# GNU Emacs Manual

# GNU Emacs Manual

Seventeenth Edition, Updated for Emacs Version 24.3.

Richard Stallman et al.

This is the Seventeenth edition of the *GNU Emacs Manual*,
updated for Emacs version 24.3.

Copyright © 1985–1987, 1993–2013 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with the Invariant Sections being "The GNU Manifesto," "Distribution" and "GNU GENERAL PUBLIC LICENSE," with the Front-Cover texts being "A GNU Manual," and with the Back-Cover Texts as in (a) below. A copy of the license is included in the section entitled "GNU Free Documentation License."

(a) The FSF's Back-Cover Text is: "You have the freedom to copy and modify this GNU manual. Buying copies from the FSF supports it in developing GNU and promoting software freedom."

Published by the Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301 USA
ISBN 978-0-9831592-4-7

Cover art by Etienne Suvasa; cover design by Matt Lee.

# Short Contents

Preface . . . . . 1
Distribution . . . . . 2
Introduction . . . . . 5
1 The Organization of the Screen . . . . . 6
2 Command Loop . . . . . 11
3 Entering and Exiting Emacs . . . . . 56
4 Basic Editing Commands . . . . . 58
5 The Minibuffer . . . . . 68
6 Running Commands by Name . . . . . 78
7 Documentation . . . . . 79
8 The Mark and the Region . . . . . 89
9 Killing and Moving Text . . . . . 95
10 Registers . . . . . 106
11 Emacs Display . . . . . 111
12 Searching and Replacement . . . . . 199
13 Commands for Fixing Typos . . . . . 217
14 Keyboard Macros . . . . . 222
15 Files . . . . . 230
16 Buffers . . . . . 272
17 Windows . . . . . 289
18 Frames . . . . . 341
19 International Character Set Support . . . . . 374
20 Major and Minor Modes . . . . . 396
21 Indentation . . . . . 451
22 Text . . . . . 454
23 Editing Programs . . . . . 514
24 Compiling and Testing Programs . . . . . 534
25 Maintaining Large Programs . . . . . 554
26 Abbrevs and Abbrev Expansion . . . . . 581
27 Dired, the Directory Editor . . . . . 588
28 The Calendar and the Diary . . . . . 604
29 Sending Mail . . . . . 624
30 Reading Mail with Rmail . . . . . 632
31 Miscellaneous Commands . . . . . 651
32 Preparing Lisp code for distribution . . . . . 681

33 Customization . . . . . 686
34 Dealing with Common Problems . . . . . 717
A GNU GENERAL PUBLIC LICENSE . . . . . 733
B GNU General Public License . . . . . 734
C GNU Free Documentation License . . . . . 745
D GNU Free Documentation License . . . . . 746
E Command Line Arguments for Emacs Invocation . . . . . 754
F X Options and Resources . . . . . 769
G Emacs 23 Antinews . . . . . 776
H Emacs and Mac OS / GNUstep . . . . . 778
I Emacs and Microsoft Windows/MS-DOS . . . . . 781
The GNU Manifesto . . . . . 791
Glossary . . . . . 799
Key (Character) Index . . . . . 822
Command and Function Index . . . . . 829
Variable Index . . . . . 845
Concept Index . . . . . 853

# Table of Contents

**Preface** ..... 1

**Distribution** ..... 2

Acknowledgments ..... 2

**Introduction** ..... 5

**1 The Organization of the Screen** ..... 6

1.1 Point ..... 6
1.2 The Echo Area ..... 7
1.3 The Mode Line ..... 8
1.4 The Menu Bar ..... 9

**2 Command Loop** ..... 11

2.1 Command Loop Overview ..... 11
2.2 Defining Commands ..... 12
2.2.1 Using `interactive` ..... 12
2.2.2 Code Characters for `interactive` ..... 14
2.2.3 Examples of Using `interactive` ..... 17
2.3 Interactive Call ..... 17
2.4 Distinguish Interactive Calls ..... 19
2.5 Information from the Command Loop ..... 20
2.6 Adjusting Point After Commands ..... 23
2.7 Input Events ..... 23
2.7.1 Keyboard Events ..... 23
2.7.2 Function Keys ..... 24
2.7.3 Mouse Events ..... 25
2.7.4 Click Events ..... 26
2.7.5 Drag Events ..... 28
2.7.6 Button-Down Events ..... 28
2.7.7 Repeat Events ..... 29
2.7.8 Motion Events ..... 30
2.7.9 Focus Events ..... 30
2.7.10 Miscellaneous System Events ..... 31
2.7.11 Event Examples ..... 32
2.7.12 Classifying Events ..... 33
2.7.13 Accessing Mouse Events ..... 35
2.7.14 Accessing Scroll Bar Events ..... 37
2.7.15 Putting Keyboard Events in Strings ..... 37
2.8 Reading Input ..... 38
2.8.1 Key Sequence Input ..... 39
2.8.2 Reading One Event ..... 41

2.8.3 Modifying and Translating Input Events ... 42
2.8.4 Invoking the Input Method ... 44
2.8.5 Quoted Character Input ... 44
2.8.6 Miscellaneous Event Input Features ... 45
2.9 Special Events ... 46
2.10 Waiting for Elapsed Time or Input ... 47
2.11 Quitting ... 48
2.12 Prefix Command Arguments ... 49
2.13 Recursive Editing ... 51
2.14 Disabling Commands ... 53
2.15 Command History ... 54
2.16 Keyboard Macros ... 54

**3 Entering and Exiting Emacs ... 56**
3.1 Entering Emacs ... 56
3.2 Exiting Emacs ... 57

**4 Basic Editing Commands ... 58**
4.1 Inserting Text ... 58
4.2 Changing the Location of Point ... 59
4.3 Erasing Text ... 61
4.4 Undoing Changes ... 62
4.5 Files ... 62
4.6 Help ... 63
4.7 Blank Lines ... 63
4.8 Continuation Lines ... 63
4.9 Cursor Position Information ... 64
4.10 Numeric Arguments ... 65
4.11 Repeating a Command ... 66

**5 The Minibuffer ... 68**
5.1 Using the Minibuffer ... 68
5.2 Minibuffers for File Names ... 68
5.3 Editing in the Minibuffer ... 69
5.4 Completion ... 70
5.4.1 Completion Example ... 70
5.4.2 Completion Commands ... 71
5.4.3 Completion Exit ... 72
5.4.4 How Completion Alternatives Are Chosen ... 72
5.4.5 Completion Options ... 74
5.5 Minibuffer History ... 74
5.6 Repeating Minibuffer Commands ... 76
5.7 Entering passwords ... 76
5.8 Yes or No Prompts ... 77

**6 Running Commands by Name ... 78**

**7 Documentation ... 79**

7.1 Documentation Basics ... 79
7.2 Access to Documentation Strings ... 80
7.3 Substituting Key Bindings in Documentation ... 82
7.4 Describing Characters for Help Messages ... 84
7.5 Help Functions ... 85

**8 The Mark and the Region ... 89**

8.1 Setting the Mark ... 89
8.2 Commands to Mark Textual Objects ... 90
8.3 Operating on the Region ... 91
8.4 The Mark Ring ... 92
8.5 The Global Mark Ring ... 93
8.6 Shift Selection ... 93
8.7 Disabling Transient Mark Mode ... 94

**9 Killing and Moving Text ... 95**

9.1 Deletion and Killing ... 95
9.1.1 Deletion ... 95
9.1.2 Killing by Lines ... 96
9.1.3 Other Kill Commands ... 97
9.1.4 Options for Killing ... 97
9.2 Yanking ... 97
9.2.1 The Kill Ring ... 98
9.2.2 Yanking Earlier Kills ... 98
9.2.3 Appending Kills ... 99
9.3 "Cut and Paste" Operations on Graphical Displays ... 100
9.3.1 Using the Clipboard ... 100
9.3.2 Cut and Paste with Other Window Applications ... 101
9.3.3 Secondary Selection ... 101
9.4 Accumulating Text ... 102
9.5 Rectangles ... 103
9.6 CUA Bindings ... 105

**10 Registers ... 106**

10.1 Saving Positions in Registers ... 106
10.2 Saving Text in Registers ... 106
10.3 Saving Rectangles in Registers ... 107
10.4 Saving Window Configurations in Registers ... 107
10.5 Keeping Numbers in Registers ... 108
10.6 Keeping File Names in Registers ... 108
10.7 Bookmarks ... 109

**11 Emacs Display ..... 111**

11.1 Refreshing the Screen ..... 111
11.2 Forcing Redisplay ..... 111
11.3 Truncation ..... 112
11.4 The Echo Area ..... 114
11.4.1 Displaying Messages in the Echo Area ..... 114
11.4.2 Reporting Operation Progress ..... 115
11.4.3 Logging Messages in '`*Messages*`' ..... 117
11.4.4 Echo Area Customization ..... 118
11.5 Reporting Warnings ..... 118
11.5.1 Warning Basics ..... 118
11.5.2 Warning Variables ..... 119
11.5.3 Warning Options ..... 120
11.5.4 Delayed Warnings ..... 121
11.6 Invisible Text ..... 121
11.7 Selective Display ..... 124
11.8 Temporary Displays ..... 125
11.9 Overlays ..... 128
11.9.1 Managing Overlays ..... 128
11.9.2 Overlay Properties ..... 131
11.9.3 Searching for Overlays ..... 134
11.10 Width ..... 135
11.11 Line Height ..... 136
11.12 Faces ..... 137
11.12.1 Face Attributes ..... 138
11.12.2 Defining Faces ..... 141
11.12.3 Face Attribute Functions ..... 143
11.12.4 Displaying Faces ..... 146
11.12.5 Face Remapping ..... 147
11.12.6 Functions for Working with Faces ..... 148
11.12.7 Automatic Face Assignment ..... 149
11.12.8 Basic Faces ..... 149
11.12.9 Font Selection ..... 150
11.12.10 Looking Up Fonts ..... 152
11.12.11 Fontsets ..... 152
11.12.12 Low-Level Font Representation ..... 154
11.13 Fringes ..... 156
11.13.1 Fringe Size and Position ..... 157
11.13.2 Fringe Indicators ..... 157
11.13.3 Fringe Cursors ..... 159
11.13.4 Fringe Bitmaps ..... 159
11.13.5 Customizing Fringe Bitmaps ..... 160
11.13.6 The Overlay Arrow ..... 161
11.14 Scroll Bars ..... 162
11.15 The `display` Property ..... 163
11.15.1 Display Specs That Replace The Text ..... 163
11.15.2 Specified Spaces ..... 164
11.15.3 Pixel Specification for Spaces ..... 165

11.15.4 Other Display Specifications ... 166
11.15.5 Displaying in the Margins ... 167
11.16 Images ... 168
11.16.1 Image Formats ... 168
11.16.2 Image Descriptors ... 169
11.16.3 XBM Images ... 172
11.16.4 XPM Images ... 172
11.16.5 GIF Images ... 173
11.16.6 TIFF Images ... 173
11.16.7 PostScript Images ... 173
11.16.8 ImageMagick Images ... 173
11.16.9 Other Image Types ... 174
11.16.10 Defining Images ... 175
11.16.11 Showing Images ... 176
11.16.12 Animated Images ... 178
11.16.13 Image Cache ... 178
11.17 Buttons ... 179
11.17.1 Button Properties ... 179
11.17.2 Button Types ... 180
11.17.3 Making Buttons ... 181
11.17.4 Manipulating Buttons ... 181
11.17.5 Button Buffer Commands ... 182
11.18 Abstract Display ... 183
11.18.1 Abstract Display Functions ... 184
11.18.2 Abstract Display Example ... 186
11.19 Blinking Parentheses ... 188
11.20 Character Display ... 189
11.20.1 Usual Display Conventions ... 189
11.20.2 Display Tables ... 190
11.20.3 Active Display Table ... 191
11.20.4 Glyphs ... 192
11.20.5 Glyphless Character Display ... 192
11.21 Beeping ... 194
11.22 Window Systems ... 194
11.23 Bidirectional Display ... 195

**12 Searching and Replacement ... 199**

12.1 Incremental Search ... 199
12.1.1 Basics of Incremental Search ... 199
12.1.2 Repeating Incremental Search ... 200
12.1.3 Errors in Incremental Search ... 200
12.1.4 Special Input for Incremental Search ... 201
12.1.5 Isearch Yanking ... 202
12.1.6 Scrolling During Incremental Search ... 202
12.1.7 Searching the Minibuffer ... 203
12.2 Nonincremental Search ... 203
12.3 Word Search ... 203
12.4 Symbol Search ... 204

12.5 Regular Expression Search ... 205
12.6 Syntax of Regular Expressions ... 206
12.7 Backslash in Regular Expressions ... 208
12.8 Regular Expression Example ... 210
12.9 Searching and Case ... 211
12.10 Replacement Commands ... 211
12.10.1 Unconditional Replacement ... 211
12.10.2 Regexp Replacement ... 212
12.10.3 Replace Commands and Case ... 213
12.10.4 Query Replace ... 213
12.11 Other Search-and-Loop Commands ... 215

**13 Commands for Fixing Typos ... 217**
13.1 Undo ... 217
13.2 Transposing Text ... 218
13.3 Case Conversion ... 219
13.4 Checking and Correcting Spelling ... 219

**14 Keyboard Macros ... 222**
14.1 Basic Use ... 222
14.2 The Keyboard Macro Ring ... 223
14.3 The Keyboard Macro Counter ... 224
14.4 Executing Macros with Variations ... 226
14.5 Naming and Saving Keyboard Macros ... 226
14.6 Editing a Keyboard Macro ... 227
14.7 Stepwise Editing a Keyboard Macro ... 228

**15 Files ... 230**
15.1 Visiting Files ... 230
15.1.1 Functions for Visiting Files ... 230
15.1.2 Subroutines of Visiting ... 233
15.2 Saving Buffers ... 234
15.3 Reading from Files ... 236
15.4 Writing to Files ... 237
15.5 File Locks ... 239
15.6 Information about Files ... 240
15.6.1 Testing Accessibility ... 240
15.6.2 Distinguishing Kinds of Files ... 242
15.6.3 Truenames ... 243
15.6.4 Other Information about Files ... 244
15.6.5 How to Locate Files in Standard Places ... 247
15.7 Changing File Names and Attributes ... 248
15.8 File Names ... 251
15.8.1 File Name Components ... 251
15.8.2 Absolute and Relative File Names ... 253
15.8.3 Directory Names ... 254
15.8.4 Functions that Expand Filenames ... 255

15.8.5 Generating Unique File Names ... 257
15.8.6 File Name Completion ... 258
15.8.7 Standard File Names ... 259
15.9 Contents of Directories ... 260
15.10 Creating, Copying and Deleting Directories ... 262
15.11 Making Certain File Names "Magic" ... 262
15.12 File Format Conversion ... 267
15.12.1 Overview ... 267
15.12.2 Round-Trip Specification ... 267
15.12.3 Piecemeal Specification ... 269

**16 Buffers ... 272**

16.1 Buffer Basics ... 272
16.2 The Current Buffer ... 272
16.3 Buffer Names ... 275
16.4 Buffer File Name ... 276
16.5 Buffer Modification ... 278
16.6 Buffer Modification Time ... 279
16.7 Read-Only Buffers ... 280
16.8 The Buffer List ... 281
16.9 Creating Buffers ... 284
16.10 Killing Buffers ... 284
16.11 Indirect Buffers ... 286
16.12 Swapping Text Between Two Buffers ... 287
16.13 The Buffer Gap ... 287

**17 Windows ... 289**

17.1 Basic Concepts of Emacs Windows ... 289
17.2 Windows and Frames ... 290
17.3 Window Sizes ... 293
17.4 Resizing Windows ... 295
17.5 Splitting Windows ... 297
17.6 Deleting Windows ... 300
17.7 Recombining Windows ... 301
17.8 Selecting Windows ... 306
17.9 Cyclic Ordering of Windows ... 307
17.10 Buffers and Windows ... 309
17.11 Switching to a Buffer in a Window ... 311
17.12 Choosing a Window for Display ... 313
17.13 Action Functions for `display-buffer` ... 314
17.14 Additional Options for Displaying Buffers ... 317
17.15 Window History ... 319
17.16 Dedicated Windows ... 321
17.17 Quitting Windows ... 321
17.18 Windows and Point ... 323
17.19 The Window Start and End Positions ... 324
17.20 Textual Scrolling ... 327
17.21 Vertical Fractional Scrolling ... 330

17.22 Horizontal Scrolling ... 331
17.23 Coordinates and Windows ... 333
17.24 Window Configurations ... 335
17.25 Window Parameters ... 337
17.26 Hooks for Window Scrolling and Changes ... 339

**18 Frames ... 341**

18.1 Creating Frames ... 342
18.2 Multiple Terminals ... 342
18.3 Frame Parameters ... 345
18.3.1 Access to Frame Parameters ... 345
18.3.2 Initial Frame Parameters ... 345
18.3.3 Window Frame Parameters ... 346
18.3.3.1 Basic Parameters ... 346
18.3.3.2 Position Parameters ... 347
18.3.3.3 Size Parameters ... 348
18.3.3.4 Layout Parameters ... 349
18.3.3.5 Buffer Parameters ... 350
18.3.3.6 Window Management Parameters ... 350
18.3.3.7 Cursor Parameters ... 351
18.3.3.8 Font and Color Parameters ... 352
18.3.4 Frame Size And Position ... 354
18.3.5 Geometry ... 355
18.4 Terminal Parameters ... 355
18.5 Frame Titles ... 356
18.6 Deleting Frames ... 357
18.7 Finding All Frames ... 357
18.8 Minibuffers and Frames ... 358
18.9 Input Focus ... 358
18.10 Visibility of Frames ... 360
18.11 Raising and Lowering Frames ... 361
18.12 Frame Configurations ... 362
18.13 Mouse Tracking ... 362
18.14 Mouse Position ... 362
18.15 Pop-Up Menus ... 363
18.16 Dialog Boxes ... 364
18.17 Pointer Shape ... 365
18.18 Window System Selections ... 366
18.19 Drag and Drop ... 367
18.20 Color Names ... 367
18.21 Text Terminal Colors ... 369
18.22 X Resources ... 369
18.23 Display Feature Testing ... 370

**19 International Character Set Support ..... 374**

19.1 Introduction to International Character Sets ..... 374
19.2 Disabling Multibyte Characters ..... 376
19.3 Language Environments ..... 377
19.4 Input Methods ..... 378
19.5 Selecting an Input Method ..... 380
19.6 Coding Systems ..... 381
19.7 Recognizing Coding Systems ..... 383
19.8 Specifying a File's Coding System ..... 385
19.9 Choosing Coding Systems for Output ..... 385
19.10 Specifying a Coding System for File Text ..... 386
19.11 Coding Systems for Interprocess Communication ..... 387
19.12 Coding Systems for File Names ..... 388
19.13 Coding Systems for Terminal I/O ..... 388
19.14 Fontsets ..... 389
19.15 Defining fontsets ..... 390
19.16 Modifying Fontsets ..... 392
19.17 Undisplayable Characters ..... 392
19.18 Unibyte Editing Mode ..... 393
19.19 Charsets ..... 394
19.20 Bidirectional Editing ..... 394

**20 Major and Minor Modes ..... 396**

20.1 Hooks ..... 396
20.1.1 Running Hooks ..... 396
20.1.2 Setting Hooks ..... 398
20.2 Major Modes ..... 399
20.2.1 Major Mode Conventions ..... 399
20.2.2 How Emacs Chooses a Major Mode ..... 403
20.2.3 Getting Help about a Major Mode ..... 405
20.2.4 Defining Derived Modes ..... 405
20.2.5 Basic Major Modes ..... 407
20.2.6 Mode Hooks ..... 408
20.2.7 Tabulated List mode ..... 409
20.2.8 Generic Modes ..... 411
20.2.9 Major Mode Examples ..... 411
20.3 Minor Modes ..... 413
20.3.1 Conventions for Writing Minor Modes ..... 413
20.3.2 Keymaps and Minor Modes ..... 415
20.3.3 Defining Minor Modes ..... 415
20.4 Mode Line Format ..... 418
20.4.1 Mode Line Basics ..... 419
20.4.2 The Data Structure of the Mode Line ..... 419
20.4.3 The Top Level of Mode Line Control ..... 421
20.4.4 Variables Used in the Mode Line ..... 422
20.4.5 %-Constructs in the Mode Line ..... 424
20.4.6 Properties in the Mode Line ..... 425
20.4.7 Window Header Lines ..... 426

20.4.8 Emulating Mode Line Formatting ..... 426
20.5 Imenu ..... 427
20.6 Font Lock Mode ..... 429
20.6.1 Font Lock Basics ..... 429
20.6.2 Search-based Fontification ..... 430
20.6.3 Customizing Search-Based Fontification ..... 434
20.6.4 Other Font Lock Variables ..... 435
20.6.5 Levels of Font Lock ..... 436
20.6.6 Precalculated Fontification ..... 436
20.6.7 Faces for Font Lock ..... 436
20.6.8 Syntactic Font Lock ..... 437
20.6.9 Multiline Font Lock Constructs ..... 438
20.6.9.1 Font Lock Multiline ..... 439
20.6.9.2 Region to Fontify after a Buffer Change ..... 440
20.7 Automatic Indentation of code ..... 440
20.7.1 Simple Minded Indentation Engine ..... 441
20.7.1.1 SMIE Setup and Features ..... 441
20.7.1.2 Operator Precedence Grammars ..... 442
20.7.1.3 Defining the Grammar of a Language ..... 443
20.7.1.4 Defining Tokens ..... 444
20.7.1.5 Living With a Weak Parser ..... 445
20.7.1.6 Specifying Indentation Rules ..... 446
20.7.1.7 Helper Functions for Indentation Rules ..... 447
20.7.1.8 Sample Indentation Rules ..... 448
20.8 Desktop Save Mode ..... 449

**21 Indentation ..... 451**

21.1 Indentation Commands ..... 451
21.2 Tab Stops ..... 452
21.3 Tabs vs. Spaces ..... 453
21.4 Convenience Features for Indentation ..... 453

**22 Text ..... 454**

22.1 Examining Text Near Point ..... 454
22.2 Examining Buffer Contents ..... 455
22.3 Comparing Text ..... 457
22.4 Inserting Text ..... 458
22.5 User-Level Insertion Commands ..... 459
22.6 Deleting Text ..... 460
22.7 User-Level Deletion Commands ..... 462
22.8 The Kill Ring ..... 464
22.8.1 Kill Ring Concepts ..... 464
22.8.2 Functions for Killing ..... 464
22.8.3 Yanking ..... 465
22.8.4 Functions for Yanking ..... 466
22.8.5 Low-Level Kill Ring ..... 467
22.8.6 Internals of the Kill Ring ..... 468
22.9 Undo ..... 469

22.10 Maintaining Undo Lists ... 471
22.11 Filling ... 473
22.12 Margins for Filling ... 475
22.13 Adaptive Fill Mode ... 477
22.14 Auto Filling ... 478
22.15 Sorting Text ... 479
22.16 Counting Columns ... 482
22.17 Indentation ... 483
22.17.1 Indentation Primitives ... 483
22.17.2 Indentation Controlled by Major Mode ... 484
22.17.3 Indenting an Entire Region ... 485
22.17.4 Indentation Relative to Previous Lines ... 486
22.17.5 Adjustable "Tab Stops" ... 487
22.17.6 Indentation-Based Motion Commands ... 487
22.18 Case Changes ... 487
22.19 Text Properties ... 489
22.19.1 Examining Text Properties ... 489
22.19.2 Changing Text Properties ... 490
22.19.3 Text Property Search Functions ... 492
22.19.4 Properties with Special Meanings ... 494
22.19.5 Formatted Text Properties ... 499
22.19.6 Stickiness of Text Properties ... 500
22.19.7 Lazy Computation of Text Properties ... 501
22.19.8 Defining Clickable Text ... 502
22.19.9 Defining and Using Fields ... 504
22.19.10 Why Text Properties are not Intervals ... 506
22.20 Substituting for a Character Code ... 507
22.21 Registers ... 507
22.22 Transposition of Text ... 509
22.23 Base 64 Encoding ... 509
22.24 Checksum/Hash ... 510
22.25 Parsing HTML and XML ... 510
22.26 Atomic Change Groups ... 511
22.27 Change Hooks ... 512

**23 Editing Programs ... 514**

23.1 Major Modes for Programming Languages ... 514
23.2 Top-Level Definitions, or Defuns ... 515
23.2.1 Left Margin Convention ... 515
23.2.2 Moving by Defuns ... 515
23.2.3 Imenu ... 516
23.2.4 Which Function Mode ... 517
23.3 Indentation for Programs ... 517
23.3.1 Basic Program Indentation Commands ... 517
23.3.2 Indenting Several Lines ... 518
23.3.3 Customizing Lisp Indentation ... 518
23.3.4 Commands for C Indentation ... 519
23.3.5 Customizing C Indentation ... 519

23.4 Commands for Editing with Parentheses ..... 520
23.4.1 Expressions with Balanced Parentheses ..... 521
23.4.2 Moving in the Parenthesis Structure ..... 522
23.4.3 Matching Parentheses ..... 522
23.5 Manipulating Comments ..... 523
23.5.1 Comment Commands ..... 523
23.5.2 Multiple Lines of Comments ..... 525
23.5.3 Options Controlling Comments ..... 525
23.6 Documentation Lookup ..... 526
23.6.1 Info Documentation Lookup ..... 526
23.6.2 Man Page Lookup ..... 526
23.6.3 Emacs Lisp Documentation Lookup ..... 527
23.7 Hideshow minor mode ..... 527
23.8 Completion for Symbol Names ..... 528
23.9 Glasses minor mode ..... 528
23.10 Semantic ..... 529
23.11 Other Features Useful for Editing Programs ..... 529
23.12 C and Related Modes ..... 530
23.12.1 C Mode Motion Commands ..... 530
23.12.2 Electric C Characters ..... 531
23.12.3 Hungry Delete Feature in C ..... 531
23.12.4 Other Commands for C Mode ..... 532
23.13 Asm Mode ..... 533

**24 Compiling and Testing Programs ..... 534**

24.1 Running Compilations under Emacs ..... 534
24.2 Compilation Mode ..... 535
24.3 Subshells for Compilation ..... 537
24.4 Searching with Grep under Emacs ..... 537
24.5 Finding Syntax Errors On The Fly ..... 539
24.6 Running Debuggers Under Emacs ..... 539
24.6.1 Starting GUD ..... 539
24.6.2 Debugger Operation ..... 540
24.6.3 Commands of GUD ..... 541
24.6.4 GUD Customization ..... 543
24.6.5 GDB Graphical Interface ..... 543
24.6.5.1 GDB User Interface Layout ..... 544
24.6.5.2 Source Buffers ..... 544
24.6.5.3 Breakpoints Buffer ..... 545
24.6.5.4 Threads Buffer ..... 545
24.6.5.5 Stack Buffer ..... 546
24.6.5.6 Other GDB Buffers ..... 546
24.6.5.7 Watch Expressions ..... 547
24.6.5.8 Multithreaded Debugging ..... 548
24.7 Executing Lisp Expressions ..... 549
24.8 Libraries of Lisp Code for Emacs ..... 549
24.9 Evaluating Emacs Lisp Expressions ..... 550
24.10 Lisp Interaction Buffers ..... 552

24.11 Running an External Lisp ..... 552

**25 Maintaining Large Programs ..... 554**

25.1 Version Control ..... 554
25.1.1 Introduction to Version Control ..... 554
25.1.1.1 Understanding the problems it addresses ..... 554
25.1.1.2 Supported Version Control Systems ..... 554
25.1.1.3 Concepts of Version Control ..... 555
25.1.1.4 Merge-based vs lock-based Version Control ..... 556
25.1.1.5 Changeset-based vs File-based Version Control ..... 556
25.1.1.6 Decentralized vs Centralized Repositories ..... 557
25.1.1.7 Types of Log File ..... 557
25.1.2 Version Control and the Mode Line ..... 557
25.1.3 Basic Editing under Version Control ..... 558
25.1.3.1 Basic Version Control with Merging ..... 558
25.1.3.2 Basic Version Control with Locking ..... 559
25.1.3.3 Advanced Control in `C-x v v` ..... 560
25.1.4 Features of the Log Entry Buffer ..... 560
25.1.5 Registering a File for Version Control ..... 561
25.1.6 Examining And Comparing Old Revisions ..... 562
25.1.7 VC Change Log ..... 563
25.1.8 Undoing Version Control Actions ..... 565
25.1.9 VC Directory Mode ..... 565
25.1.9.1 The VC Directory Buffer ..... 565
25.1.9.2 VC Directory Commands ..... 566
25.1.10 Version Control Branches ..... 567
25.1.10.1 Switching between Branches ..... 568
25.1.10.2 Pulling Changes into a Branch ..... 568
25.1.10.3 Merging Branches ..... 569
25.1.10.4 Creating New Branches ..... 569
25.2 Change Logs ..... 570
25.2.1 Change Log Commands ..... 570
25.2.2 Format of ChangeLog ..... 571
25.3 Tags Tables ..... 571
25.3.1 Source File Tag Syntax ..... 572
25.3.2 Creating Tags Tables ..... 574
25.3.3 Etags Regexps ..... 575
25.3.4 Selecting a Tags Table ..... 576
25.3.5 Finding a Tag ..... 577
25.3.6 Searching and Replacing with Tags Tables ..... 578
25.3.7 Tags Table Inquiries ..... 579
25.4 Emacs Development Environment ..... 580

**26 Abbrevs and Abbrev Expansion ..... 581**

26.1 Abbrev Tables ..... 581
26.2 Defining Abbrevs ..... 582
26.3 Saving Abbrevs in Files ..... 583
26.4 Looking Up and Expanding Abbreviations ..... 584
26.5 Standard Abbrev Tables ..... 586
26.6 Abbrev Properties ..... 586
26.7 Abbrev Table Properties ..... 587

**27 Dired, the Directory Editor ..... 588**

27.1 Entering Dired ..... 588
27.2 Navigation in the Dired Buffer ..... 589
27.3 Deleting Files with Dired ..... 589
27.4 Flagging Many Files at Once ..... 590
27.5 Visiting Files in Dired ..... 591
27.6 Dired Marks vs. Flags ..... 591
27.7 Operating on Files ..... 593
27.8 Shell Commands in Dired ..... 595
27.9 Transforming File Names in Dired ..... 596
27.10 File Comparison with Dired ..... 597
27.11 Subdirectories in Dired ..... 598
27.12 Moving Over Subdirectories ..... 598
27.13 Hiding Subdirectories ..... 599
27.14 Updating the Dired Buffer ..... 599
27.15 Dired and `find` ..... 600
27.16 Editing the Dired Buffer ..... 601
27.17 Viewing Image Thumbnails in Dired ..... 601
27.18 Other Dired Features ..... 602

**28 The Calendar and the Diary ..... 604**

28.1 Movement in the Calendar ..... 604
28.1.1 Motion by Standard Lengths of Time ..... 604
28.1.2 Beginning or End of Week, Month or Year ..... 605
28.1.3 Specified Dates ..... 605
28.2 Scrolling in the Calendar ..... 606
28.3 Counting Days ..... 606
28.4 Miscellaneous Calendar Commands ..... 606
28.5 Writing Calendar Files ..... 607
28.6 Holidays ..... 608
28.7 Times of Sunrise and Sunset ..... 609
28.8 Phases of the Moon ..... 610
28.9 Conversion To and From Other Calendars ..... 611
28.9.1 Supported Calendar Systems ..... 611
28.9.2 Converting To Other Calendars ..... 612
28.9.3 Converting From Other Calendars ..... 613
28.9.4 Converting from the Mayan Calendar ..... 614
28.10 The Diary ..... 615

28.10.1 Displaying the Diary ... 615
28.10.2 The Diary File ... 616
28.10.3 Date Formats ... 617
28.10.4 Commands to Add to the Diary ... 618
28.10.5 Special Diary Entries ... 618
28.11 Appointments ... 620
28.12 Importing and Exporting Diary Entries ... 621
28.13 Daylight Saving Time ... 621
28.14 Summing Time Intervals ... 622

**29 Sending Mail ... 624**

29.1 The Format of the Mail Buffer ... 624
29.2 Mail Header Fields ... 625
29.3 Mail Aliases ... 626
29.4 Mail Commands ... 627
29.4.1 Mail Sending ... 627
29.4.2 Mail Header Editing ... 628
29.4.3 Citing Mail ... 629
29.4.4 Mail Miscellany ... 630
29.5 Mail Signature ... 630
29.6 Mail Amusements ... 631
29.7 Mail-Composition Methods ... 631

**30 Reading Mail with Rmail ... 632**

30.1 Basic Concepts of Rmail ... 632
30.2 Scrolling Within a Message ... 632
30.3 Moving Among Messages ... 633
30.4 Deleting Messages ... 634
30.5 Rmail Files and Inboxes ... 635
30.6 Multiple Rmail Files ... 636
30.7 Copying Messages Out to Files ... 637
30.8 Labels ... 638
30.9 Rmail Attributes ... 639
30.10 Sending Replies ... 640
30.11 Summaries ... 642
30.11.1 Making Summaries ... 642
30.11.2 Editing in Summaries ... 643
30.12 Sorting the Rmail File ... 645
30.13 Display of Messages ... 645
30.14 Rmail and Coding Systems ... 647
30.15 Editing Within a Message ... 647
30.16 Digest Messages ... 648
30.17 Reading Rot13 Messages ... 648
30.18 `movemail` program ... 648
30.19 Retrieving Mail from Remote Mailboxes ... 649
30.20 Retrieving Mail from Local Mailboxes in Various Formats ... 650

**31 Miscellaneous Commands ..... 651**
31.1 Gnus ..... 651
31.1.1 Gnus Buffers ..... 651
31.1.2 When Gnus Starts Up ..... 651
31.1.3 Using the Gnus Group Buffer ..... 652
31.1.4 Using the Gnus Summary Buffer ..... 652
31.2 Document Viewing ..... 653
31.2.1 DocView Navigation ..... 653
31.2.2 DocView Searching ..... 654
31.2.3 DocView Slicing ..... 654
31.2.4 DocView Conversion ..... 654
31.3 Running Shell Commands from Emacs ..... 655
31.3.1 Single Shell Commands ..... 655
31.3.2 Interactive Subshell ..... 656
31.3.3 Shell Mode ..... 657
31.3.4 Shell Prompts ..... 659
31.3.5 Shell Command History ..... 660
31.3.5.1 Shell History Ring ..... 660
31.3.5.2 Shell History Copying ..... 661
31.3.5.3 Shell History References ..... 661
31.3.6 Directory Tracking ..... 662
31.3.7 Shell Mode Options ..... 662
31.3.8 Emacs Terminal Emulator ..... 663
31.3.9 Term Mode ..... 664
31.3.10 Remote Host Shell ..... 664
31.3.11 Serial Terminal ..... 665
31.4 Using Emacs as a Server ..... 665
31.4.1 Invoking `emacsclient` ..... 666
31.4.2 `emacsclient` Options ..... 667
31.5 Printing Hard Copies ..... 669
31.5.1 PostScript Hardcopy ..... 670
31.5.2 Variables for PostScript Hardcopy ..... 671
31.5.3 Printing Package ..... 672
31.6 Sorting Text ..... 672
31.7 Editing Binary Files ..... 674
31.8 Saving Emacs Sessions ..... 675
31.9 Recursive Editing Levels ..... 675
31.10 Emulation ..... 676
31.11 Hyperlinking and Navigation Features ..... 677
31.11.1 Following URLs ..... 677
31.11.2 Activating URLs ..... 678
31.11.3 Finding Files and URLs at Point ..... 678
31.12 Other Amusements ..... 679

**32 Preparing Lisp code for distribution ..... 681**

32.1 Packaging Basics ..... 681
32.2 Simple Packages ..... 682
32.3 Multi-file Packages ..... 683
32.4 Creating and Maintaining Package Archives ..... 684

**33 Customization ..... 686**

33.1 Easy Customization Interface ..... 686
33.1.1 Customization Groups ..... 686
33.1.2 Browsing and Searching for Settings ..... 687
33.1.3 Changing a Variable ..... 687
33.1.4 Saving Customizations ..... 690
33.1.5 Customizing Faces ..... 690
33.1.6 Customizing Specific Items ..... 691
33.1.7 Custom Themes ..... 692
33.1.8 Creating Custom Themes ..... 693
33.2 Variables ..... 694
33.2.1 Examining and Setting Variables ..... 695
33.2.2 Hooks ..... 696
33.2.3 Local Variables ..... 697
33.2.4 Local Variables in Files ..... 698
33.2.4.1 Specifying File Variables ..... 698
33.2.4.2 Safety of File Variables ..... 700
33.2.5 Per-Directory Local Variables ..... 701
33.3 Customizing Key Bindings ..... 702
33.3.1 Keymaps ..... 703
33.3.2 Prefix Keymaps ..... 703
33.3.3 Local Keymaps ..... 704
33.3.4 Minibuffer Keymaps ..... 704
33.3.5 Changing Key Bindings Interactively ..... 705
33.3.6 Rebinding Keys in Your Init File ..... 706
33.3.7 Modifier Keys ..... 707
33.3.8 Rebinding Function Keys ..... 707
33.3.9 Named ASCII Control Characters ..... 708
33.3.10 Rebinding Mouse Buttons ..... 709
33.3.11 Disabling Commands ..... 711
33.4 The Emacs Initialization File ..... 711
33.4.1 Init File Syntax ..... 712
33.4.2 Init File Examples ..... 713
33.4.3 Terminal-specific Initialization ..... 715
33.4.4 How Emacs Finds Your Init File ..... 716
33.4.5 Non-ASCII Characters in Init Files ..... 716

**34 Dealing with Common Problems ..... 717**
34.1 Quitting and Aborting ..... 717
34.2 Dealing with Emacs Trouble ..... 718
34.2.1 If DEL Fails to Delete ..... 718
34.2.2 Recursive Editing Levels ..... 719
34.2.3 Garbage on the Screen ..... 719
34.2.4 Garbage in the Text ..... 720
34.2.5 Running out of Memory ..... 720
34.2.6 When Emacs Crashes ..... 720
34.2.7 Recovery After a Crash ..... 721
34.2.8 Emergency Escape ..... 722
34.3 Reporting Bugs ..... 722
34.3.1 Reading Existing Bug Reports and Known Problems ... 722
34.3.2 When Is There a Bug ..... 723
34.3.3 Understanding Bug Reporting ..... 724
34.3.4 Checklist for Bug Reports ..... 725
34.3.5 Sending Patches for GNU Emacs ..... 730
34.4 Contributing to Emacs Development ..... 731
34.5 How To Get Help with GNU Emacs ..... 731

**Appendix A GNU GENERAL PUBLIC LICENSE ..... 733**

**Appendix B GNU General Public License ... 734**

**Appendix C GNU Free Documentation License ..... 745**

**Appendix D GNU Free Documentation License ..... 746**

**Appendix E Command Line Arguments for Emacs Invocation ..... 754**
E.1 Action Arguments ..... 754
E.2 Initial Options ..... 755
E.3 Command Argument Example ..... 758
E.4 Environment Variables ..... 758
E.4.1 General Variables ..... 758
E.4.2 Miscellaneous Variables ..... 761
E.4.3 The MS-Windows System Registry ..... 762
E.5 Specifying the Display Name ..... 762
E.6 Font Specification Options ..... 763
E.7 Window Color Options ..... 763
E.8 Options for Window Size and Position ..... 765
E.9 Internal and External Borders ..... 766

E.10 Frame Titles ..... 767
E.11 Icons ..... 767
E.12 Other Display Options ..... 767

**Appendix F X Options and Resources ..... 769**

F.1 X Resources ..... 769
F.2 Table of X Resources for Emacs ..... 770
F.3 GTK resources ..... 771
F.3.1 GTK Resource Basics ..... 772
F.3.2 GTK widget names ..... 772
F.3.3 GTK Widget Names in Emacs ..... 773
F.3.4 GTK styles ..... 774

**Appendix G Emacs 23 Antinews ..... 776**

G.1 Old Lisp Features in Emacs 23 ..... 776

**Appendix H Emacs and Mac OS / GNUstep ..... 778**

H.1 Basic Emacs usage under Mac OS and GNUstep ..... 778
H.1.1 Grabbing environment variables ..... 778
H.2 Mac / GNUstep Customization ..... 779
H.2.1 Font and Color Panels ..... 779
H.2.2 Customization options specific to Mac OS / GNUstep ..... 779
H.3 Windowing System Events under Mac OS / GNUstep ..... 779
H.4 GNUstep Support ..... 780

**Appendix I Emacs and Microsoft Windows/MS-DOS ..... 781**

I.1 How to Start Emacs on MS-Windows ..... 781
I.2 Text Files and Binary Files ..... 782
I.3 File Names on MS-Windows ..... 783
I.4 Emulation of `ls` on MS-Windows ..... 784
I.5 HOME and Startup Directories on MS-Windows ..... 784
I.6 Keyboard Usage on MS-Windows ..... 785
I.7 Mouse Usage on MS-Windows ..... 785
I.8 Subprocesses on Windows 9X/ME and Windows NT/2K/XP ..... 786
I.9 Printing and MS-Windows ..... 787
I.10 Specifying Fonts on MS-Windows ..... 788
I.11 Miscellaneous Windows-specific features ..... 790

**The GNU Manifesto** ..... **791**
What's GNU? Gnu's Not Unix! ..... 791
Why I Must Write GNU ..... 792
Why GNU Will Be Compatible with Unix ..... 792
How GNU Will Be Available ..... 792
Why Many Other Programmers Want to Help ..... 792
How You Can Contribute ..... 793
Why All Computer Users Will Benefit ..... 793
Some Easily Rebutted Objections to GNU's Goals ..... 794

**Glossary** ..... **799**

**Key (Character) Index** ..... **822**

**Command and Function Index** ..... **829**

**Variable Index** ..... **845**

**Concept Index** ..... **853**

# Preface

This manual documents the use and simple customization of the Emacs editor. Simple Emacs customizations do not require you to be a programmer, but if you are not interested in customizing, you can ignore the customization hints.

This is primarily a reference manual, but can also be used as a primer. If you are new to Emacs, we recommend you start with the integrated, learn-by-doing tutorial, before reading the manual. To run the tutorial, start Emacs and type `C-h t`. The tutorial describes commands, tells you when to try them, and explains the results. The tutorial is available in several languages.

On first reading, just skim chapters 1 and 2, which describe the notational conventions of the manual and the general appearance of the Emacs display screen. Note which questions are answered in these chapters, so you can refer back later. After reading chapter 4, you should practice the commands shown there. The next few chapters describe fundamental techniques and concepts that are used constantly. You need to understand them thoroughly, so experiment with them until you are fluent.

Chapters 14 through 19 describe intermediate-level features that are useful for many kinds of editing. Chapter 20 and following chapters describe optional but useful features; read those chapters when you need them.

Read the Common Problems chapter if Emacs does not seem to be working properly. It explains how to cope with several common problems (see Section 34.2 [Dealing with Emacs Trouble], page 718), as well as when and how to report Emacs bugs (see Section 34.3 [Bugs], page 722).

To find the documentation of a particular command, look in the index. Keys (character commands) and command names have separate indexes. There is also a glossary, with a cross reference for each term.

This manual is available as a printed book and also as an Info file. The Info file is for reading from Emacs itself, or with the Info program. Info is the principal format for documentation in the GNU system. The Info file and the printed book contain substantially the same text and are generated from the same source files, which are also distributed with GNU Emacs.

GNU Emacs is a member of the Emacs editor family. There are many Emacs editors, all sharing common principles of organization. For information on the underlying philosophy of Emacs and the lessons learned from its development, see *Emacs, the Extensible, Customizable Self-Documenting Display Editor*, available from `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-519A.pdf`.

This version of the manual is mainly intended for use with GNU Emacs installed on GNU and Unix systems. GNU Emacs can also be used on MS-DOS, Microsoft Windows, and Macintosh systems. The Info file version of this manual contains some more information about using Emacs on those systems. Those systems use different file name syntax; in addition MS-DOS does not support all GNU Emacs features. See Appendix I [Microsoft Windows], page 781, for information about using Emacs on Windows. See Appendix H [Mac OS / GNUstep], page 778, for information about using Emacs on Macintosh (and GNUstep).

# Distribution

GNU Emacs is *free software*; this means that everyone is free to use it and free to redistribute it under certain conditions. GNU Emacs is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of GNU Emacs that they might get from you. The precise conditions are found in the GNU General Public License that comes with Emacs and also appears in this manual[1]. See Appendix A [Copying], page 733.

One way to get a copy of GNU Emacs is from someone else who has it. You need not ask for our permission to do so, or tell any one else; just copy it. If you have access to the Internet, you can get the latest distribution version of GNU Emacs by anonymous FTP; see `http://www.gnu.org/software/emacs` on our website for more information.

You may also receive GNU Emacs when you buy a computer. Computer manufacturers are free to distribute copies on the same terms that apply to everyone else. These terms require them to give you the full sources, including whatever changes they may have made, and to permit you to redistribute the GNU Emacs received from them under the usual terms of the General Public License. In other words, the program must be free for you when you get it, not just free for the manufacturer.

If you find GNU Emacs useful, please **send a donation** to the Free Software Foundation to support our work. Donations to the Free Software Foundation are tax deductible in the US. If you use GNU Emacs at your workplace, please suggest that the company make a donation. For more information on how you can help, see `http://www.gnu.org/help/help.html`.

We also sell hardcopy versions of this manual and *An Introduction to Programming in Emacs Lisp*, by Robert J. Chassell. You can visit our online store at `http://shop.fsf.org/`. The income from sales goes to support the foundation's purpose: the development of new free software, and improvements to our existing programs including GNU Emacs.

If you need to contact the Free Software Foundation, see `http://www.fsf.org/about/contact/`, or write to

Free Software Foundation
51 Franklin Street, Fifth Floor
Boston, MA 02110-1301
USA

## Acknowledgments

Contributors to GNU Emacs include Jari Aalto, Per Abrahamsen, Tomas Abrahamsson, Jay K. Adams, Alon Albert, Michael Albinus, Nagy Andras, Benjamin Andresen, Ralf Angeli, Dmitry Antipov, Joe Arceneaux, Emil Åström, Miles Bader, David Bakhash, Juanma Barranquero, Eli Barzilay, Thomas Baumann, Steven L. Baur, Jay Belanger, Alexander L. Belikoff, Thomas Bellman, Scott Bender, Boaz Ben-Zvi, Sergey Berezin, Karl Berry,

[1] This manual is itself covered by the GNU Free Documentation License. This license is similar in spirit to the General Public License, but is more suitable for documentation. See Appendix D [GNU Free Documentation License], page 746.

Anna M. Bigatti, Ray Blaak, Martin Blais, Jim Blandy, Johan Bockgård, Jan Böcker, Joel Boehland, Lennart Borgman, Per Bothner, Terrence Brannon, Frank Bresz, Peter Breton, Emmanuel Briot, Kevin Broadey, Vincent Broman, Michael Brouwer, David M. Brown, Stefan Bruda, Georges Brun-Cottan, Joe Buehler, Scott Byer, Włodek Bzyl, Bill Carpenter, Per Cederqvist, Hans Chalupsky, Chris Chase, Bob Chassell, Andrew Choi, Chong Yidong, Sacha Chua, Stewart Clamen, James Clark, Mike Clarkson, Glynn Clements, Andrew Cohen, Daniel Colascione, Edward O'Connor, Christoph Conrad, Ludovic Courtès, Andrew Csillag, Toby Cubitt, Baoqiu Cui, Doug Cutting, Mathias Dahl, Julien Danjou, Satyaki Das, Vivek Dasmohapatra, Dan Davison, Michael DeCorte, Gary Delp, Nachum Dershowitz, Dave Detlefs, Matthieu Devin, Christophe de Dinechin, Eri Ding, Jan Djärv, Lawrence R. Dodd, Carsten Dominik, Scott Draves, Benjamin Drieu, Viktor Dukhovni, Jacques Duthen, Dmitry Dzhus, John Eaton, Rolf Ebert, Carl Edman, David Edmondson, Paul Eggert, Stephen Eglen, Christian Egli, Torbjörn Einarsson, Tsugutomo Enami, David Engster, Hans Henrik Eriksen, Michael Ernst, Ata Etemadi, Frederick Farnbach, Oscar Figueiredo, Fred Fish, Steve Fisk, Karl Fogel, Gary Foster, Eric S. Fraga, Romain Francoise, Noah Friedman, Andreas Fuchs, Shigeru Fukaya, Hallvard Furuseth, Keith Gabryelski, Peter S. Galbraith, Kevin Gallagher, Fabián E. Gallina, Kevin Gallo, Juan León Lahoz García, Howard Gayle, Daniel German, Stephen Gildea, Julien Gilles, David Gillespie, Bob Glickstein, Deepak Goel, David De La Harpe Golden, Boris Goldowsky, David Goodger, Chris Gray, Kevin Greiner, Michelangelo Grigni, Odd Gripenstam, Kai Großjohann, Michael Gschwind, Bastien Guerry, Henry Guillaume, Doug Gwyn, Bruno Haible, Ken'ichi Handa, Lars Hansen, Chris Hanson, Jesper Harder, Alexandru Harsanyi, K. Shane Hartman, John Heidemann, Jon K. Hellan, Magnus Henoch, Markus Heritsch, Dirk Herrmann, Karl Heuer, Manabu Higashida, Konrad Hinsen, Anders Holst, Jeffrey C. Honig, Tassilo Horn, Kurt Hornik, Tom Houlder, Joakim Hove, Denis Howe, Lars Ingebrigtsen, Andrew Innes, Seiichiro Inoue, Philip Jackson, Martyn Jago, Pavel Janik, Paul Jarc, Ulf Jasper, Thorsten Jolitz, Michael K. Johnson, Kyle Jones, Terry Jones, Simon Josefsson, Alexandre Julliard, Arne Jørgensen, Tomoji Kagatani, Brewster Kahle, Tokuya Kameshima, Lute Kamstra, Ivan Kanis, David Kastrup, David Kaufman, Henry Kautz, Taichi Kawabata, Taro Kawagishi, Howard Kaye, Michael Kifer, Richard King, Peter Kleiweg, Karel Klíč, Shuhei Kobayashi, Pavel Kobyakov, Larry K. Kolodney, David M. Koppelman, Koseki Yoshinori, Robert Krawitz, Sebastian Kremer, Ryszard Kubiak, Igor Kuzmin, David Kågedal, Daniel LaLiberte, Karl Landstrom, Mario Lang, Aaron Larson, James R. Larus, Vinicius Jose Latorre, Werner Lemberg, Frederic Lepied, Peter Liljenberg, Christian Limpach, Lars Lindberg, Chris Lindblad, Anders Lindgren, Thomas Link, Juri Linkov, Francis Litterio, Sergey Litvinov, Emilio C. Lopes, Martin Lorentzon, Dave Love, Eric Ludlam, Károly Lőrentey, Sascha Lüdecke, Greg McGary, Roland McGrath, Michael McNamara, Alan Mackenzie, Christopher J. Madsen, Neil M. Mager, Ken Manheimer, Bill Mann, Brian Marick, Simon Marshall, Bengt Martensson, Charlie Martin, Yukihiro Matsumoto, Tomohiro Matsuyama, David Maus, Thomas May, Will Mengarini, David Megginson, Stefan Merten, Ben A. Mesander, Wayne Mesard, Brad Miller, Lawrence Mitchell, Richard Mlynarik, Gerd Moellmann, Stefan Monnier, Keith Moore, Jan Moringen, Morioka Tomohiko, Glenn Morris, Don Morrison, Diane Murray, Riccardo Murri, Sen Nagata, Erik Naggum, Gergely Nagy, Nobuyoshi Nakada, Thomas Neumann, Mike Newton, Thien-Thi Nguyen, Jurgen Nickelsen, Dan Nicolaescu, Hrvoje Niksic, Jeff Norden, Andrew Norman, Kentaro Ohkouchi, Christian Ohler, Kenichi Okada, Alexandre Oliva, Bob Olson, Michael Olson, Takaaki Ota, Pieter E. J. Pareit, Ross Patterson, David Pearson, Juan Pechiar,

Jeff Peck, Damon Anton Permezel, Tom Perrine, William M. Perry, Per Persson, Jens Petersen, Daniel Pfeiffer, Justus Piater, Richard L. Pieri, Fred Pierresteguy, François Pinard, Daniel Pittman, Christian Plaunt, Alexander Pohoyda, David Ponce, Francesco A. Potorti, Michael D. Prange, Mukesh Prasad, Ken Raeburn, Marko Rahamaa, Ashwin Ram, Eric S. Raymond, Paul Reilly, Edward M. Reingold, David Reitter, Alex Rezinsky, Rob Riepel, Lara Rios, Adrian Robert, Nick Roberts, Roland B. Roberts, John Robinson, Denis B. Roegel, Danny Roozendaal, Sebastian Rose, William Rosenblatt, Markus Rost, Guillermo J. Rozas, Martin Rudalics, Ivar Rummelhoff, Jason Rumney, Wolfgang Rupprecht, Benjamin Rutt, Kevin Ryde, James B. Salem, Masahiko Sato, Timo Savola, Jorgen Schaefer, Holger Schauer, William Schelter, Ralph Schleicher, Gregor Schmid, Michael Schmidt, Ronald S. Schnell, Philippe Schnoebelen, Jan Schormann, Alex Schroeder, Stefan Schoef, Rainer Schoepf, Raymond Scholz, Eric Schulte, Andreas Schwab, Randal Schwartz, Oliver Seidel, Manuel Serrano, Paul Sexton, Hovav Shacham, Stanislav Shalunov, Marc Shapiro, Richard Sharman, Olin Shivers, Tibor Šimko, Espen Skoglund, Rick Sladkey, Lynn Slater, Chris Smith, David Smith, Paul D. Smith, Wilson Snyder, William Sommerfeld, Simon South, Andre Spiegel, Michael Staats, Thomas Steffen, Ulf Stegemann, Reiner Steib, Sam Steingold, Ake Stenhoff, Peter Stephenson, Ken Stevens, Andy Stewart, Jonathan Stigelman, Martin Stjernholm, Kim F. Storm, Steve Strassmann, Christopher Suckling, Olaf Sylvester, Naoto Takahashi, Steven Tamm, Luc Teirlinck, Jean-Philippe Theberge, Jens T. Berger Thielemann, Spencer Thomas, Jim Thompson, Toru Tomabechi, David O'Toole, Markus Triska, Tom Tromey, Enami Tsugutomo, Eli Tziperman, Daiki Ueno, Masanobu Umeda, Rajesh Vaidheeswarran, Neil W. Van Dyke, Didier Verna, Joakim Verona, Ulrik Vieth, Geoffrey Voelker, Johan Vromans, Inge Wallin, John Paul Wallington, Colin Walters, Barry Warsaw, Christoph Wedler, Ilja Weis, Zhang Weize, Morten Welinder, Joseph Brian Wells, Rodney Whitby, John Wiegley, Sascha Wilde, Ed Wilkinson, Mike Williams, Roland Winkler, Bill Wohler, Steven A. Wood, Dale R. Worley, Francis J. Wright, Felix S. T. Wu, Tom Wurgler, Yamamoto Mitsuharu, Katsumi Yamaoka, Masatake Yamato, Jonathan Yavner, Ryan Yeske, Ilya Zakharevich, Milan Zamazal, Victor Zandy, Eli Zaretskii, Jamie Zawinski, Andrew Zhilin, Shenghuo Zhu, Piotr Zielinski, Ian T. Zimmermann, Reto Zimmermann, Neal Ziring, Teodor Zlatanov, and Detlev Zundel.

# Introduction

You are reading about GNU Emacs, the GNU incarnation of the advanced, self-documenting, customizable, extensible editor Emacs. (The 'G' in 'GNU' is not silent.)

We call Emacs *advanced* because it can do much more than simple insertion and deletion of text. It can control subprocesses, indent programs automatically, show multiple files at once, and more. Emacs editing commands operate in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in various programming languages.

*Self-documenting* means that at any time you can use special commands, known as *help commands*, to find out what your options are, or to find out what any command does, or to find all the commands that pertain to a given topic. See ⟨undefined⟩ [Help], page ⟨undefined⟩.

*Customizable* means that you can easily alter the behavior of Emacs commands in simple ways. For instance, if you use a programming language in which comments start with '<**' and end with '**>', you can tell the Emacs comment manipulation commands to use those strings (see Section 23.5 [Comments], page 523). To take another example, you can rebind the basic cursor motion commands (up, down, left and right) to any keys on the keyboard that you find comfortable. See Chapter 33 [Customization], page 686.

*Extensible* means that you can go beyond simple customization and create entirely new commands. New commands are simply programs written in the Lisp language, which are run by Emacs's own Lisp interpreter. Existing commands can even be redefined in the middle of an editing session, without having to restart Emacs. Most of the editing commands in Emacs are written in Lisp; the few exceptions could have been written in Lisp but use C instead for efficiency. Writing an extension is programming, but non-programmers can use it afterwards. See Section "Preface" in *An Introduction to Programming in Emacs Lisp*, if you want to learn Emacs Lisp programming.

# 1 The Organization of the Screen

On a graphical display, such as on GNU/Linux using the X Window System, Emacs occupies a "graphical window". On a text terminal, Emacs occupies the entire terminal screen. We will use the term *frame* to mean a graphical window or terminal screen occupied by Emacs. Emacs behaves very similarly on both kinds of frames. It normally starts out with just one frame, but you can create additional frames if you wish (see Chapter 18 [Frames], page 341).

Each frame consists of several distinct regions. At the top of the frame is a *menu bar*, which allows you to access commands via a series of menus. On a graphical display, directly below the menu bar is a *tool bar*, a row of icons that perform editing commands if you click on them. At the very bottom of the frame is an *echo area*, where informative messages are displayed and where you enter information when Emacs asks for it.

The main area of the frame, below the tool bar (if one exists) and above the echo area, is called *the window*. Henceforth in this manual, we will use the word "window" in this sense. Graphical display systems commonly use the word "window" with a different meaning; but, as stated above, we refer to those "graphical windows" as "frames".

An Emacs window is where the *buffer*—the text you are editing—is displayed. On a graphical display, the window possesses a *scroll bar* on one side, which can be used to scroll through the buffer. The last line of the window is a *mode line*. This displays various information about what is going on in the buffer, such as whether there are unsaved changes, the editing modes that are in use, the current line number, and so forth.

When you start Emacs, there is normally only one window in the frame. However, you can subdivide this window horizontally or vertically to create multiple windows, each of which can independently display a buffer (see Chapter 17 [Windows], page 289).

At any time, one window is the *selected window*. On a graphical display, the selected window shows a more prominent cursor (usually solid and blinking); other windows show a less prominent cursor (usually a hollow box). On a text terminal, there is only one cursor, which is shown in the selected window. The buffer displayed in the selected window is called the *current buffer*, and it is where editing happens. Most Emacs commands implicitly apply to the current buffer; the text displayed in unselected windows is mostly visible for reference. If you use multiple frames on a graphical display, selecting a particular frame selects a window in that frame.

## 1.1 Point

The cursor in the selected window shows the location where most editing commands take effect, which is called *point*[1]. Many Emacs commands move point to different places in the buffer; for example, you can place point by clicking mouse button 1 (normally the left button) at the desired location.

By default, the cursor in the selected window is drawn as a solid block and appears to be *on* a character, but you should think of point as *between* two characters; it is situated *before* the character under the cursor. For example, if your text looks like '`frob`' with the cursor over the '`b`', then point is between the '`o`' and the '`b`'. If you insert the character '`!`'

[1] The term "point" comes from the character '`.`', which was the command in TECO (the language in which the original Emacs was written) for accessing the editing position.

at that position, the result is ‘fro!b’, with point between the ‘!’ and the ‘b’. Thus, the cursor remains over the ‘b’, as before.

If you are editing several files in Emacs, each in its own buffer, each buffer has its own value of point. A buffer that is not currently displayed remembers its value of point if you later display it again. Furthermore, if a buffer is displayed in multiple windows, each of those windows has its own value of point.

See ⟨undefined⟩ [Cursor Display], page ⟨undefined⟩, for options that control how Emacs displays the cursor.

## 1.2 The Echo Area

The line at the very bottom of the frame is the *echo area*. It is used to display small amounts of text for various purposes.

The echo area is so-named because one of the things it is used for is *echoing*, which means displaying the characters of a multi-character command as you type. Single-character commands are not echoed. Multi-character commands (see ⟨undefined⟩ [Keys], page ⟨undefined⟩) are echoed if you pause for more than a second in the middle of a command. Emacs then echoes all the characters of the command so far, to prompt you for the rest. Once echoing has started, the rest of the command echoes immediately as you type it. This behavior is designed to give confident users fast response, while giving hesitant users maximum feedback.

The echo area is also used to display an *error message* when a command cannot do its job. Error messages may be accompanied by beeping or by flashing the screen.

Some commands display informative messages in the echo area to tell you what the command has done, or to provide you with some specific information. These *informative* messages, unlike error messages, are not accompanied with a beep or flash. For example, C-x = (hold down CTRL and type x, then let go of CTRL and type =) displays a message describing the character at point, its position in the buffer, and its current column in the window. Commands that take a long time often display messages ending in ‘...’ while they are working (sometimes also indicating how much progress has been made, as a percentage), and add ‘done’ when they are finished.

Informative echo area messages are saved in a special buffer named ‘*Messages*’. (We have not explained buffers yet; see Chapter 16 [Buffers], page 272, for more information about them.) If you miss a message that appeared briefly on the screen, you can switch to the ‘*Messages*’ buffer to see it again. The ‘*Messages*’ buffer is limited to a certain number of lines, specified by the variable message-log-max. (We have not explained variables either; see Section 33.2 [Variables], page 694, for more information about them.) Beyond this limit, one line is deleted from the beginning whenever a new message line is added at the end.

See ⟨undefined⟩ [Display Custom], page ⟨undefined⟩, for options that control how Emacs uses the echo area.

The echo area is also used to display the *minibuffer*, a special window where you can input arguments to commands, such as the name of a file to be edited. When the minibuffer is in use, the text displayed in the echo area begins with a *prompt string*, and the active cursor appears within the minibuffer, which is temporarily considered the selected window. You can always get out of the minibuffer by typing C-g. See Chapter 5 [Minibuffer], page 68.

## 1.3 The Mode Line

At the bottom of each window is a *mode line*, which describes what is going on in the current buffer. When there is only one window, the mode line appears right above the echo area; it is the next-to-last line in the frame. On a graphical display, the mode line is drawn with a 3D box appearance. Emacs also usually draws the mode line of the selected window with a different color than that of unselected windows, in order to make it stand out.

The text displayed in the mode line has the following format:

```
cs:ch-fr  buf      pos line   (major minor)
```

On a text terminal, this text is followed by a series of dashes extending to the right edge of the window. These dashes are omitted on a graphical display.

The *cs* string and the colon character after it describe the character set and newline convention used for the current buffer. Normally, Emacs automatically handles these settings for you, but it is sometimes useful to have this information.

*cs* describes the character set of the text in the buffer (see Section 19.6 [Coding Systems], page 381). If it is a dash (‘-’), that indicates no special character set handling (with the possible exception of end-of-line conventions, described in the next paragraph). ‘=’ means no conversion whatsoever, and is usually used for files containing non-textual data. Other characters represent various *coding systems*—for example, ‘1’ represents ISO Latin-1.

On a text terminal, *cs* is preceded by two additional characters that describe the coding systems for keyboard input and terminal output. Furthermore, if you are using an input method, *cs* is preceded by a string that identifies the input method (see Section 19.4 [Input Methods], page 378).

The character after *cs* is usually a colon. If a different string is displayed, that indicates a nontrivial end-of-line convention for encoding a file. Usually, lines of text are separated by *newline characters* in a file, but two other conventions are sometimes used. The MS-DOS convention uses a “carriage-return” character followed by a “linefeed” character; when editing such files, the colon changes to either a backslash (‘\’) or ‘(DOS)’, depending on the operating system. Another convention, employed by older Macintosh systems, uses a “carriage-return” character instead of a newline; when editing such files, the colon changes to either a forward slash (‘/’) or ‘(Mac)’. On some systems, Emacs displays ‘(Unix)’ instead of the colon for files that use newline as the line separator.

The next element on the mode line is the string indicated by *ch*. This shows two dashes (‘--’) if the buffer displayed in the window has the same contents as the corresponding file on the disk; i.e., if the buffer is “unmodified”. If the buffer is modified, it shows two stars (‘**’). For a read-only buffer, it shows ‘%*’ if the buffer is modified, and ‘%%’ otherwise.

The character after *ch* is normally a dash (‘-’). However, if the default-directory for the current buffer is on a remote machine, ‘@’ is displayed instead (see Section 15.8 [File Names], page 251).

*fr* gives the selected frame name (see Chapter 18 [Frames], page 341). It appears only on text terminals. The initial frame’s name is ‘F1’.

*buf* is the name of the buffer displayed in the window. Usually, this is the same as the name of a file you are editing. See Chapter 16 [Buffers], page 272.

*pos* tells you whether there is additional text above the top of the window, or below the bottom. If your buffer is small and all of it is visible in the window, *pos* is ‘All’. Otherwise,

it is '`Top`' if you are looking at the beginning of the buffer, '`Bot`' if you are looking at the end of the buffer, or '*nn*`%`', where *nn* is the percentage of the buffer above the top of the window. With Size Indication mode, you can display the size of the buffer as well. See ⟨undefined⟩ [Optional Mode Line], page ⟨undefined⟩.

*line* is the character '`L`' followed by the line number at point. (You can display the current column number too, by turning on Column Number mode. See ⟨undefined⟩ [Optional Mode Line], page ⟨undefined⟩.)

*major* is the name of the *major mode* used in the buffer. A major mode is a principal editing mode for the buffer, such as Text mode, Lisp mode, C mode, and so forth. See Section 20.2 [Major Modes], page 399. Some major modes display additional information after the major mode name. For example, Compilation buffers and Shell buffers display the status of the subprocess.

*minor* is a list of some of the enabled *minor modes*, which are optional editing modes that provide additional features on top of the major mode. See Section 20.3 [Minor Modes], page 413.

Some features are listed together with the minor modes whenever they are turned on, even though they are not really minor modes. '`Narrow`' means that the buffer being displayed has editing restricted to only a portion of its text (see ⟨undefined⟩ [Narrowing], page ⟨undefined⟩). '`Def`' means that a keyboard macro is currently being defined (see Chapter 14 [Keyboard Macros], page 222).

In addition, if Emacs is inside a recursive editing level, square brackets ('`[...]`') appear around the parentheses that surround the modes. If Emacs is in one recursive editing level within another, double square brackets appear, and so on. Since recursive editing levels affect Emacs globally, such square brackets appear in the mode line of every window. See Section 31.9 [Recursive Edit], page 675.

You can change the appearance of the mode line as well as the format of its contents. See ⟨undefined⟩ [Optional Mode Line], page ⟨undefined⟩. In addition, the mode line is mouse-sensitive; clicking on different parts of the mode line performs various commands. See ⟨undefined⟩ [Mode Line Mouse], page ⟨undefined⟩.

## 1.4 The Menu Bar

Each Emacs frame normally has a *menu bar* at the top which you can use to perform common operations. There's no need to list them here, as you can more easily see them yourself.

On a graphical display, you can use the mouse to choose a command from the menu bar. An arrow on the right edge of a menu item means it leads to a subsidiary menu, or *submenu*. A '`...`' at the end of a menu item means that the command will prompt you for further input before it actually does anything.

Some of the commands in the menu bar have ordinary key bindings as well; if so, a key binding is shown in parentheses after the item itself. To view the full command name and documentation for a menu item, type `C-h k`, and then select the menu bar with the mouse in the usual way (see ⟨undefined⟩ [Key Help], page ⟨undefined⟩).

Instead of using the mouse, you can also invoke the first menu bar item by pressing F10 (to run the command `menu-bar-open`). You can then navigate the menus with the arrow keys. To activate a selected menu item, press RET; to cancel menu navigation, press ESC.

On a text terminal, you can use the menu bar by typing `M-‘` or F10 (these run the command `tmm-menubar`). This lets you select a menu item with the keyboard. A provisional choice appears in the echo area. You can use the up and down arrow keys to move through the menu to different items, and then you can type RET to select the item. Each menu item is also designated by a letter or digit (usually the initial of some word in the item's name). This letter or digit is separated from the item name by '`==>`'. You can type the item's letter or digit to select the item.

# 2 Command Loop

When you run Emacs, it enters the *editor command loop* almost immediately. This loop reads key sequences, executes their definitions, and displays the results. In this chapter, we describe how these things are done, and the subroutines that allow Lisp programs to do them.

## 2.1 Command Loop Overview

The first thing the command loop must do is read a key sequence, which is a sequence of input events that translates into a command. It does this by calling the function `read-key-sequence`. Lisp programs can also call this function (see Section 2.8.1 [Key Sequence Input], page 39). They can also read input at a lower level with `read-key` or `read-event` (see Section 2.8.2 [Reading One Event], page 41), or discard pending input with `discard-input` (see Section 2.8.6 [Event Input Misc], page 45).

The key sequence is translated into a command through the currently active keymaps. See ⟨undefined⟩ [Key Lookup], page ⟨undefined⟩, for information on how this is done. The result should be a keyboard macro or an interactively callable function. If the key is `M-x`, then it reads the name of another command, which it then calls. This is done by the command `execute-extended-command` (see Section 2.3 [Interactive Call], page 17).

Prior to executing the command, Emacs runs `undo-boundary` to create an undo boundary. See Section 22.10 [Maintaining Undo], page 471.

To execute a command, Emacs first reads its arguments by calling `command-execute` (see Section 2.3 [Interactive Call], page 17). For commands written in Lisp, the `interactive` specification says how to read the arguments. This may use the prefix argument (see Section 2.12 [Prefix Command Arguments], page 49) or may read with prompting in the minibuffer (see ⟨undefined⟩ [Minibuffers], page ⟨undefined⟩). For example, the command `find-file` has an `interactive` specification which says to read a file name using the minibuffer. The function body of `find-file` does not use the minibuffer, so if you call `find-file` as a function from Lisp code, you must supply the file name string as an ordinary Lisp function argument.

If the command is a keyboard macro (i.e., a string or vector), Emacs executes it using `execute-kbd-macro` (see Chapter 14 [Keyboard Macros], page 222).

`pre-command-hook` [Variable]

This normal hook is run by the editor command loop before it executes each command. At that time, `this-command` contains the command that is about to run, and `last-command` describes the previous command. See Section 2.5 [Command Loop Info], page 20.

`post-command-hook` [Variable]

This normal hook is run by the editor command loop after it executes each command (including commands terminated prematurely by quitting or by errors). At that time, `this-command` refers to the command that just ran, and `last-command` refers to the command before that.

This hook is also run when Emacs first enters the command loop (at which point `this-command` and `last-command` are both `nil`).

Quitting is suppressed while running `pre-command-hook` and `post-command-hook`. If an error happens while executing one of these hooks, it does not terminate execution of the hook; instead the error is silenced and the function in which the error occurred is removed from the hook.

A request coming into the Emacs server (see Section "Emacs Server" in *The GNU Emacs Manual*) runs these two hooks just as a keyboard command does.

## 2.2 Defining Commands

The special form `interactive` turns a Lisp function into a command. The `interactive` form must be located at top-level in the function body (usually as the first form in the body), or in the `interactive-form` property of the function symbol. When the `interactive` form is located in the function body, it does nothing when actually executed. Its presence serves as a flag, which tells the Emacs command loop that the function can be called interactively. The argument of the `interactive` form controls the reading of arguments for an interactive call.

### 2.2.1 Using `interactive`

This section describes how to write the `interactive` form that makes a Lisp function an interactively-callable command, and how to examine a command's `interactive` form.

`interactive` *arg-descriptor* [Special Form]

This special form declares that a function is a command, and that it may therefore be called interactively (via `M-x` or by entering a key sequence bound to it). The argument *arg-descriptor* declares how to compute the arguments to the command when the command is called interactively.

A command may be called from Lisp programs like any other function, but then the caller supplies the arguments and *arg-descriptor* has no effect.

The `interactive` form must be located at top-level in the function body, or in the function symbol's `interactive-form` property (see ⟨undefined⟩ [Symbol Properties], page ⟨undefined⟩). It has its effect because the command loop looks for it before calling the function (see Section 2.3 [Interactive Call], page 17). Once the function is called, all its body forms are executed; at this time, if the `interactive` form occurs within the body, the form simply returns `nil` without even evaluating its argument.

By convention, you should put the `interactive` form in the function body, as the first top-level form. If there is an `interactive` form in both the `interactive-form` symbol property and the function body, the former takes precedence. The `interactive-form` symbol property can be used to add an interactive form to an existing function, or change how its arguments are processed interactively, without redefining the function.

There are three possibilities for the argument *arg-descriptor*:

- It may be omitted or `nil`; then the command is called with no arguments. This leads quickly to an error if the command requires one or more arguments.
- It may be a string; its contents are a sequence of elements separated by newlines, one for each argument[1]. Each element consists of a code character (see Section 2.2.2 [Interactive

[1] Some elements actually supply two arguments.

Codes], page 14) optionally followed by a prompt (which some code characters use and some ignore). Here is an example:

```
(interactive "P\nbFrobnicate buffer: ")
```

The code letter '`P`' sets the command's first argument to the raw command prefix (see Section 2.12 [Prefix Command Arguments], page 49). '`bFrobnicate buffer: `' prompts the user with '`Frobnicate buffer: `' to enter the name of an existing buffer, which becomes the second and final argument.

The prompt string can use '`%`' to include previous argument values (starting with the first argument) in the prompt. This is done using `format` (see ⟨undefined⟩ [Formatting Strings], page ⟨undefined⟩). For example, here is how you could read the name of an existing buffer followed by a new name to give to that buffer:

```
(interactive "bBuffer to rename: \nsRename buffer %s to: ")
```

If '`*`' appears at the beginning of the string, then an error is signaled if the buffer is read-only.

If '`@`' appears at the beginning of the string, and if the key sequence used to invoke the command includes any mouse events, then the window associated with the first of those events is selected before the command is run.

If '`^`' appears at the beginning of the string, and if the command was invoked through *shift-translation*, set the mark and activate the region temporarily, or extend an already active region, before the command is run. If the command was invoked without shift-translation, and the region is temporarily active, deactivate the region before the command is run. Shift-translation is controlled on the user level by `shift-select-mode`; see Section "Shift Selection" in *The GNU Emacs Manual*.

You can use '`*`', '`@`', and `^` together; the order does not matter. Actual reading of arguments is controlled by the rest of the prompt string (starting with the first character that is not '`*`', '`@`', or '`^`').

- It may be a Lisp expression that is not a string; then it should be a form that is evaluated to get a list of arguments to pass to the command. Usually this form will call various functions to read input from the user, most often through the minibuffer (see ⟨undefined⟩ [Minibuffers], page ⟨undefined⟩) or directly from the keyboard (see Section 2.8 [Reading Input], page 38).

  Providing point or the mark as an argument value is also common, but if you do this *and* read input (whether using the minibuffer or not), be sure to get the integer values of point or the mark after reading. The current buffer may be receiving subprocess output; if subprocess output arrives while the command is waiting for input, it could relocate point and the mark.

  Here's an example of what *not* to do:

```
(interactive
 (list (region-beginning) (region-end)
       (read-string "Foo: " nil 'my-history)))
```

  Here's how to avoid the problem, by examining point and the mark after reading the keyboard input:

```
(interactive
 (let ((string (read-string "Foo: " nil 'my-history)))
   (list (region-beginning) (region-end) string)))
```

**Warning:** the argument values should not include any data types that can't be printed and then read. Some facilities save `command-history` in a file to be read in the subsequent sessions; if a command's arguments contain a data type that prints using '`#<...>`' syntax, those facilities won't work.

There are, however, a few exceptions: it is ok to use a limited set of expressions such as `(point)`, `(mark)`, `(region-beginning)`, and `(region-end)`, because Emacs recognizes them specially and puts the expression (rather than its value) into the command history. To see whether the expression you wrote is one of these exceptions, run the command, then examine `(car command-history)`.

`interactive-form` *function* [Function]

This function returns the `interactive` form of *function*. If *function* is an interactively callable function (see Section 2.3 [Interactive Call], page 17), the value is the command's `interactive` form `(interactive` *spec*`)`, which specifies how to compute its arguments. Otherwise, the value is `nil`. If *function* is a symbol, its function definition is used.

### 2.2.2 Code Characters for `interactive`

The code character descriptions below contain a number of key words, defined here as follows:

**Completion**
: Provide completion. TAB, SPC, and RET perform name completion because the argument is read using `completing-read` (see Section 5.4 [Completion], page 70). `?` displays a list of possible completions.

**Existing**
: Require the name of an existing object. An invalid name is not accepted; the commands to exit the minibuffer do not exit if the current input is not valid.

**Default**
: A default value of some sort is used if the user enters no text in the minibuffer. The default depends on the code character.

**No I/O**
: This code letter computes an argument without reading any input. Therefore, it does not use a prompt string, and any prompt string you supply is ignored.

  Even though the code letter doesn't use a prompt string, you must follow it with a newline if it is not the last code character in the string.

**Prompt**
: A prompt immediately follows the code character. The prompt ends either with the end of the string or with a newline.

**Special**
: This code character is meaningful only at the beginning of the interactive string, and it does not look for a prompt or a newline. It is a single, isolated character.

Here are the code character descriptions for use with `interactive`:

'`*`'
: Signal an error if the current buffer is read-only. Special.

'`@`'
: Select the window mentioned in the first mouse event in the key sequence that invoked this command. Special.

'`^`'
: If the command was invoked through shift-translation, set the mark and activate the region temporarily, or extend an already active region, before the command

is run. If the command was invoked without shift-translation, and the region is temporarily active, deactivate the region before the command is run. Special.

'a' A function name (i.e., a symbol satisfying `fboundp`). Existing, Completion, Prompt.

'b' The name of an existing buffer. By default, uses the name of the current buffer (see Chapter 16 [Buffers], page 272). Existing, Completion, Default, Prompt.

'B' A buffer name. The buffer need not exist. By default, uses the name of a recently used buffer other than the current buffer. Completion, Default, Prompt.

'c' A character. The cursor does not move into the echo area. Prompt.

'C' A command name (i.e., a symbol satisfying `commandp`). Existing, Completion, Prompt.

'd' The position of point, as an integer (see Section 1.1 [Point], page 6). No I/O.

'D' A directory name. The default is the current default directory of the current buffer, `default-directory` (see Section 15.8.4 [File Name Expansion], page 255). Existing, Completion, Default, Prompt.

'e' The first or next non-keyboard event in the key sequence that invoked the command. More precisely, 'e' gets events that are lists, so you can look at the data in the lists. See Section 2.7 [Input Events], page 23. No I/O.

You use 'e' for mouse events and for special system events (see Section 2.7.10 [Misc Events], page 31). The event list that the command receives depends on the event. See Section 2.7 [Input Events], page 23, which describes the forms of the list for each event in the corresponding subsections.

You can use 'e' more than once in a single command's interactive specification. If the key sequence that invoked the command has *n* events that are lists, the *n*th 'e' provides the *n*th such event. Events that are not lists, such as function keys and ASCII characters, do not count where 'e' is concerned.

'f' A file name of an existing file (see Section 15.8 [File Names], page 251). The default directory is `default-directory`. Existing, Completion, Default, Prompt.

'F' A file name. The file need not exist. Completion, Default, Prompt.

'G' A file name. The file need not exist. If the user enters just a directory name, then the value is just that directory name, with no file name within the directory added. Completion, Default, Prompt.

'i' An irrelevant argument. This code always supplies `nil` as the argument's value. No I/O.

'k' A key sequence (see ⟨undefined⟩ [Key Sequences], page ⟨undefined⟩). This keeps reading events until a command (or undefined command) is found in the current key maps. The key sequence argument is represented as a string or vector. The cursor does not move into the echo area. Prompt.

If 'k' reads a key sequence that ends with a down-event, it also reads and discards the following up-event. You can get access to that up-event with the 'U' code character.

This kind of input is used by commands such as `describe-key` and `global-set-key`.

'K' A key sequence, whose definition you intend to change. This works like 'k', except that it suppresses, for the last input event in the key sequence, the conversions that are normally used (when necessary) to convert an undefined key into a defined one.

'm' The position of the mark, as an integer. No I/O.

'M' Arbitrary text, read in the minibuffer using the current buffer's input method, and returned as a string (see Section "Input Methods" in *The GNU Emacs Manual*). Prompt.

'n' A number, read with the minibuffer. If the input is not a number, the user has to try again. 'n' never uses the prefix argument. Prompt.

'N' The numeric prefix argument; but if there is no prefix argument, read a number as with `n`. The value is always a number. See Section 2.12 [Prefix Command Arguments], page 49. Prompt.

'p' The numeric prefix argument. (Note that this 'p' is lower case.) No I/O.

'P' The raw prefix argument. (Note that this 'P' is upper case.) No I/O.

'r' Point and the mark, as two numeric arguments, smallest first. This is the only code letter that specifies two successive arguments rather than one. No I/O.

's' Arbitrary text, read in the minibuffer and returned as a string (see ⟨undefined⟩ [Text from Minibuffer], page ⟨undefined⟩). Terminate the input with either `C-j` or RET. (`C-q` may be used to include either of these characters in the input.) Prompt.

'S' An interned symbol whose name is read in the minibuffer. Terminate the input with either `C-j` or RET. Other characters that normally terminate a symbol (e.g., whitespace, parentheses and brackets) do not do so here. Prompt.

'U' A key sequence or `nil`. Can be used after a 'k' or 'K' argument to get the up-event that was discarded (if any) after 'k' or 'K' read a down-event. If no up-event has been discarded, 'U' provides `nil` as the argument. No I/O.

'v' A variable declared to be a user option (i.e., satisfying the predicate `custom-variable-p`). This reads the variable using `read-variable`. See ⟨undefined⟩ [Definition of read-variable], page ⟨undefined⟩. Existing, Completion, Prompt.

'x' A Lisp object, specified with its read syntax, terminated with a `C-j` or RET. The object is not evaluated. See ⟨undefined⟩ [Object from Minibuffer], page ⟨undefined⟩. Prompt.

'X' A Lisp form's value. 'X' reads as 'x' does, then evaluates the form so that its value becomes the argument for the command. Prompt.

'z' A coding system name (a symbol). If the user enters null input, the argument value is `nil`. See Section 19.6 [Coding Systems], page 381. Completion, Existing, Prompt.

‘Z’ A coding system name (a symbol)—but only if this command has a prefix argument. With no prefix argument, ‘Z’ provides `nil` as the argument value. Completion, Existing, Prompt.

### 2.2.3 Examples of Using `interactive`

Here are some examples of `interactive`:

```
(defun foo1 ()              ; foo1 takes no arguments,
    (interactive)           ;    just moves forward two words.
    (forward-word 2))
     ⇒ foo1

(defun foo2 (n)             ; foo2 takes one argument,
    (interactive "^p")      ;    which is the numeric prefix.
                            ; under shift-select-mode,
                            ;    will activate or extend region.
    (forward-word (* 2 n)))
     ⇒ foo2

(defun foo3 (n)             ; foo3 takes one argument,
    (interactive "nCount:") ;    which is read with the Minibuffer.
    (forward-word (* 2 n)))
     ⇒ foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
     ⇒ three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     ⇒ nil
```

## 2.3 Interactive Call

After the command loop has translated a key sequence into a command, it invokes that command using the function `command-execute`. If the command is a function, `command-execute` calls `call-interactively`, which reads the arguments and calls the command. You can also call these functions yourself.

Note that the term "command", in this context, refers to an interactively callable function (or function-like object), or a keyboard macro. It does not refer to the key sequence used to invoke a command (see Section 33.3.1 [Keymaps], page 703).

`commandp` *object* **&optional** *for-call-interactively* [Function]

This function returns `t` if *object* is a command. Otherwise, it returns `nil`.

Commands include strings and vectors (which are treated as keyboard macros), lambda expressions that contain a top-level `interactive` form (see Section 2.2.1 [Using Interactive], page 12), byte-code function objects made from such lambda expressions, autoload objects that are declared as interactive (non-`nil` fourth argument to `autoload`), and some primitive functions. Also, a symbol is considered a command if it has a non-`nil` `interactive-form` property, or if its function definition satisfies `commandp`.

If *for-call-interactively* is non-`nil`, then `commandp` returns `t` only for objects that `call-interactively` could call—thus, not for keyboard macros.

See `documentation` in Section 7.2 [Accessing Documentation], page 80, for a realistic example of using `commandp`.

`call-interactively` *command* **&optional** *record-flag keys* [Function]

This function calls the interactively callable function *command*, providing arguments according to its interactive calling specifications. It returns whatever *command* returns.

If, for instance, you have a function with the following signature:

```
(defun foo (begin end)
  (interactive "r")
  ...)
```

then saying

```
(call-interactively 'foo)
```

will call `foo` with the region (`point` and `mark`) as the arguments.

An error is signaled if *command* is not a function or if it cannot be called interactively (i.e., is not a command). Note that keyboard macros (strings and vectors) are not accepted, even though they are considered commands, because they are not functions. If *command* is a symbol, then `call-interactively` uses its function definition.

If *record-flag* is non-`nil`, then this command and its arguments are unconditionally added to the list `command-history`. Otherwise, the command is added only if it uses the minibuffer to read an argument. See Section 2.15 [Command History], page 54.

The argument *keys*, if given, should be a vector which specifies the sequence of events to supply if the command inquires which events were used to invoke it. If *keys* is omitted or `nil`, the default is the return value of `this-command-keys-vector`. See [Definition of this-command-keys-vector], page 22.

`command-execute` *command* **&optional** *record-flag keys special* [Function]

This function executes *command*. The argument *command* must satisfy the `commandp` predicate; i.e., it must be an interactively callable function or a keyboard macro.

A string or vector as *command* is executed with `execute-kbd-macro`. A function is passed to `call-interactively` (see above), along with the *record-flag* and *keys* arguments.

If *command* is a symbol, its function definition is used in its place. A symbol with an `autoload` definition counts as a command if it was declared to stand for an interactively callable function. Such a definition is handled by loading the specified library and then rechecking the definition of the symbol.

The argument *special*, if given, means to ignore the prefix argument and not clear it. This is used for executing special events (see Section 2.9 [Special Events], page 46).

`execute-extended-command` *prefix-argument* [Command]

This function reads a command name from the minibuffer using `completing-read` (see Section 5.4 [Completion], page 70). Then it uses `command-execute` to call the specified command. Whatever that command returns becomes the value of `execute-extended-command`.

If the command asks for a prefix argument, it receives the value *prefix-argument*. If `execute-extended-command` is called interactively, the current raw prefix argument is used for *prefix-argument*, and thus passed on to whatever command is run.

`execute-extended-command` is the normal definition of `M-x`, so it uses the string '`M-x` ' as a prompt. (It would be better to take the prompt from the events used to invoke `execute-extended-command`, but that is painful to implement.) A description of the value of the prefix argument, if any, also becomes part of the prompt.

```
(execute-extended-command 3)
---------- Buffer: Minibuffer ----------
3 M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

## 2.4 Distinguish Interactive Calls

Sometimes a command should display additional visual feedback (such as an informative message in the echo area) for interactive calls only. There are three ways to do this. The recommended way to test whether the function was called using `call-interactively` is to give it an optional argument `print-message` and use the `interactive` spec to make it non-`nil` in interactive calls. Here's an example:

```
(defun foo (&optional print-message)
  (interactive "p")
  (when print-message
    (message "foo")))
```

We use `"p"` because the numeric prefix argument is never `nil`. Defined in this way, the function does display the message when called from a keyboard macro.

The above method with the additional argument is usually best, because it allows callers to say "treat this call as interactive". But you can also do the job by testing `called-interactively-p`.

`called-interactively-p` *kind* [Function]

This function returns `t` when the calling function was called using `call-interactively`.

The argument *kind* should be either the symbol `interactive` or the symbol `any`. If it is `interactive`, then `called-interactively-p` returns `t` only if the call was made directly by the user—e.g., if the user typed a key sequence bound to the calling function, but *not* if the user ran a keyboard macro that called the function (see Chapter 14 [Keyboard Macros], page 222). If *kind* is `any`, `called-interactively-p` returns `t` for any kind of interactive call, including keyboard macros.

If in doubt, use `any`; the only known proper use of `interactive` is if you need to decide whether to display a helpful message while a function is running.

A function is never considered to be called interactively if it was called via Lisp evaluation (or with `apply` or `funcall`).

Here is an example of using `called-interactively-p`:

```
(defun foo ()
  (interactive)
  (when (called-interactively-p 'any)
    (message "Interactive!")
    'foo-called-interactively))

;; Type M-x foo.
     ⊣ Interactive!

(foo)
     ⇒ nil
```

Here is another example that contrasts direct and indirect calls to `called-interactively-p`.

```
(defun bar ()
  (interactive)
  (message "%s" (list (foo) (called-interactively-p 'any))))

;; Type M-x bar.
     ⊣ (nil t)
```

## 2.5 Information from the Command Loop

The editor command loop sets several Lisp variables to keep status records for itself and for commands that are run. With the exception of `this-command` and `last-command` it's generally a bad idea to change any of these variables in a Lisp program.

`last-command` [Variable]

This variable records the name of the previous command executed by the command loop (the one before the current command). Normally the value is a symbol with a function definition, but this is not guaranteed.

The value is copied from `this-command` when a command returns to the command loop, except when the command has specified a prefix argument for the following command.

This variable is always local to the current terminal and cannot be buffer-local. See Section 18.2 [Multiple Terminals], page 342.

`real-last-command` [Variable]
This variable is set up by Emacs just like `last-command`, but never altered by Lisp programs.

`last-repeatable-command` [Variable]
This variable stores the most recently executed command that was not part of an input event. This is the command `repeat` will try to repeat, See Section "Repeating" in *The GNU Emacs Manual*.

`this-command` [Variable]
This variable records the name of the command now being executed by the editor command loop. Like `last-command`, it is normally a symbol with a function definition.

The command loop sets this variable just before running a command, and copies its value into `last-command` when the command finishes (unless the command specified a prefix argument for the following command).

Some commands set this variable during their execution, as a flag for whatever command runs next. In particular, the functions for killing text set `this-command` to `kill-region` so that any kill commands immediately following will know to append the killed text to the previous kill.

If you do not want a particular command to be recognized as the previous command in the case where it got an error, you must code that command to prevent this. One way is to set `this-command` to `t` at the beginning of the command, and set `this-command` back to its proper value at the end, like this:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ...do the work...
    (setq this-command old-this-command)))
```

We do not bind `this-command` with `let` because that would restore the old value in case of error—a feature of `let` which in this case does precisely what we want to avoid.

`this-original-command` [Variable]
This has the same value as `this-command` except when command remapping occurs (see ⟨undefined⟩ [Remapping Commands], page ⟨undefined⟩). In that case, `this-command` gives the command actually run (the result of remapping), and `this-original-command` gives the command that was specified to run but remapped into another command.

`this-command-keys` [Function]

This function returns a string or vector containing the key sequence that invoked the present command, plus any previous commands that generated the prefix argument for this command. Any events read by the command using `read-event` without a timeout get tacked on to the end.

However, if the command has called `read-key-sequence`, it returns the last read key sequence. See Section 2.8.1 [Key Sequence Input], page 39. The value is a string if all events in the sequence were characters that fit in a string. See Section 2.7 [Input Events], page 23.

```
(this-command-keys)
;; Now use C-u C-x C-e to evaluate that.
     ⇒ "^U^X^E"
```

`this-command-keys-vector` [Function]

Like `this-command-keys`, except that it always returns the events in a vector, so you don't need to deal with the complexities of storing input events in a string (see Section 2.7.15 [Strings of Events], page 37).

`clear-this-command-keys` **&optional** *keep-record* [Function]

This function empties out the table of events for `this-command-keys` to return. Unless *keep-record* is non-`nil`, it also empties the records that the function `recent-keys` (see ⟨undefined⟩ [Recording Input], page ⟨undefined⟩) will subsequently return. This is useful after reading a password, to prevent the password from echoing inadvertently as part of the next command in certain cases.

`last-nonmenu-event` [Variable]

This variable holds the last input event read as part of a key sequence, not counting events resulting from mouse menus.

One use of this variable is for telling `x-popup-menu` where to pop up a menu. It is also used internally by `y-or-n-p` (see ⟨undefined⟩ [Yes-or-No Queries], page ⟨undefined⟩).

`last-command-event` [Variable]

This variable is set to the last input event that was read by the command loop as part of a command. The principal use of this variable is in `self-insert-command`, which uses it to decide which character to insert.

```
last-command-event
;; Now use C-u C-x C-e to evaluate that.
     ⇒ 5
```

The value is 5 because that is the ASCII code for `C-e`.

`last-event-frame` [Variable]

This variable records which frame the last input event was directed to. Usually this is the frame that was selected when the event was generated, but if that frame has redirected input focus to another frame, the value is the frame to which the event was redirected. See Section 18.9 [Input Focus], page 358.

If the last event came from a keyboard macro, the value is `macro`.

## 2.6 Adjusting Point After Commands

It is not easy to display a value of point in the middle of a sequence of text that has the `display`, `composition` or is invisible. Therefore, after a command finishes and returns to the command loop, if point is within such a sequence, the command loop normally moves point to the edge of the sequence.

A command can inhibit this feature by setting the variable `disable-point-adjustment`:

`disable-point-adjustment` [Variable]

If this variable is non-`nil` when a command returns to the command loop, then the command loop does not check for those text properties, and does not move point out of sequences that have them.

The command loop sets this variable to `nil` before each command, so if a command sets it, the effect applies only to that command.

`global-disable-point-adjustment` [Variable]

If you set this variable to a non-`nil` value, the feature of moving point out of these sequences is completely turned off.

## 2.7 Input Events

The Emacs command loop reads a sequence of *input events* that represent keyboard or mouse activity, or system events sent to Emacs. The events for keyboard activity are characters or symbols; other events are always lists. This section describes the representation and meaning of input events in detail.

`eventp` *object* [Function]

This function returns non-`nil` if *object* is an input event or event type.

Note that any symbol might be used as an event or an event type. `eventp` cannot distinguish whether a symbol is intended by Lisp code to be used as an event. Instead, it distinguishes whether the symbol has actually been used in an event that has been read as input in the current Emacs session. If a symbol has not yet been so used, `eventp` returns `nil`.

### 2.7.1 Keyboard Events

There are two kinds of input you can get from the keyboard: ordinary keys, and function keys. Ordinary keys correspond to characters; the events they generate are represented in Lisp as characters. The event type of a character event is the character itself (an integer); see Section 2.7.12 [Classifying Events], page 33.

An input character event consists of a *basic code* between 0 and 524287, plus any or all of these *modifier bits*:

meta The $2^{27}$ bit in the character code indicates a character typed with the meta key held down.

control The $2^{26}$ bit in the character code indicates a non-ASCII control character.

ASCII control characters such as `C-a` have special basic codes of their own, so Emacs needs no special bit to indicate them. Thus, the code for `C-a` is just 1.

But if you type a control combination not in ASCII, such as `%` with the control key, the numeric value you get is the code for `%` plus $2^{26}$ (assuming the terminal supports non-ASCII control characters).

shift
: The $2^{25}$ bit in the character code indicates an ASCII control character typed with the shift key held down.

  For letters, the basic code itself indicates upper versus lower case; for digits and punctuation, the shift key selects an entirely different character with a different basic code. In order to keep within the ASCII character set whenever possible, Emacs avoids using the $2^{25}$ bit for those characters.

  However, ASCII provides no way to distinguish `C-A` from `C-a`, so Emacs uses the $2^{25}$ bit in `C-A` and not in `C-a`.

hyper
: The $2^{24}$ bit in the character code indicates a character typed with the hyper key held down.

super
: The $2^{23}$ bit in the character code indicates a character typed with the super key held down.

alt
: The $2^{22}$ bit in the character code indicates a character typed with the alt key held down. (The key labeled ALT on most keyboards is actually treated as the meta key, not this.)

It is best to avoid mentioning specific bit numbers in your program. To test the modifier bits of a character, use the function `event-modifiers` (see Section 2.7.12 [Classifying Events], page 33). When making key bindings, you can use the read syntax for characters with modifier bits (‘`\C-`’, ‘`\M-`’, and so on). For making key bindings with `define-key`, you can use lists such as `(control hyper ?x)` to specify the characters (see ⟨undefined⟩ [Changing Key Bindings], page ⟨undefined⟩). The function `event-convert-list` converts such a list into an event type (see Section 2.7.12 [Classifying Events], page 33).

### 2.7.2 Function Keys

Most keyboards also have *function keys*—keys that have names or symbols that are not characters. Function keys are represented in Emacs Lisp as symbols; the symbol's name is the function key's label, in lower case. For example, pressing a key labeled F1 generates an input event represented by the symbol `f1`.

The event type of a function key event is the event symbol itself. See Section 2.7.12 [Classifying Events], page 33.

Here are a few special cases in the symbol-naming convention for function keys:

`backspace`, `tab`, `newline`, `return`, `delete`
: These keys correspond to common ASCII control characters that have special keys on most keyboards.

  In ASCII, `C-i` and TAB are the same character. If the terminal can distinguish between them, Emacs conveys the distinction to Lisp programs by representing the former as the integer 9, and the latter as the symbol `tab`.

  Most of the time, it's not useful to distinguish the two. So normally `local-function-key-map` (see ⟨undefined⟩ [Translation Keymaps], page ⟨undefined⟩) is set up to map `tab` into 9. Thus, a key binding for character code 9 (the

character `C-i`) also applies to `tab`. Likewise for the other symbols in this group. The function `read-char` likewise converts these events into characters.

In ASCII, BS is really `C-h`. But `backspace` converts into the character code 127 (DEL), not into code 8 (BS). This is what most users prefer.

`left`, `up`, `right`, `down`
: Cursor arrow keys

`kp-add`, `kp-decimal`, `kp-divide`, ...
: Keypad keys (to the right of the regular keyboard).

`kp-0`, `kp-1`, ...
: Keypad keys with digits.

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`
: Keypad PF keys.

`kp-home`, `kp-left`, `kp-up`, `kp-right`, `kp-down`
: Keypad arrow keys. Emacs normally translates these into the corresponding non-keypad keys `home`, `left`, ...

`kp-prior`, `kp-next`, `kp-end`, `kp-begin`, `kp-insert`, `kp-delete`
: Additional keypad duplicates of keys ordinarily found elsewhere. Emacs normally translates these into the like-named non-keypad keys.

You can use the modifier keys ALT, CTRL, HYPER, META, SHIFT, and SUPER with function keys. The way to represent them is with prefixes in the symbol name:

'`A-`' The alt modifier.

'`C-`' The control modifier.

'`H-`' The hyper modifier.

'`M-`' The meta modifier.

'`S-`' The shift modifier.

'`s-`' The super modifier.

Thus, the symbol for the key F3 with META held down is `M-f3`. When you use more than one prefix, we recommend you write them in alphabetical order; but the order does not matter in arguments to the key-binding lookup and modification functions.

### 2.7.3 Mouse Events

Emacs supports four kinds of mouse events: click events, drag events, button-down events, and motion events. All mouse events are represented as lists. The CAR of the list is the event type; this says which mouse button was involved, and which modifier keys were used with it. The event type can also distinguish double or triple button presses (see Section 2.7.7 [Repeat Events], page 29). The rest of the list elements give position and time information.

For key lookup, only the event type matters: two events of the same type necessarily run the same command. The command can access the full values of these events using the '`e`' interactive code. See Section 2.2.2 [Interactive Codes], page 14.

A key sequence that starts with a mouse event is read using the keymaps of the buffer in the window that the mouse was in, not the current buffer. This does not imply that clicking in a window selects that window or its buffer—that is entirely under the control of the command binding of the key sequence.

### 2.7.4 Click Events

When the user presses a mouse button and releases it at the same location, that generates a *click* event. All mouse click event share the same format:

```
(event-type position click-count)
```

*event-type* This is a symbol that indicates which mouse button was used. It is one of the symbols `mouse-1`, `mouse-2`, ..., where the buttons are numbered left to right.

You can also use prefixes '`A-`', '`C-`', '`H-`', '`M-`', '`S-`' and '`s-`' for modifiers alt, control, hyper, meta, shift and super, just as you would with function keys.

This symbol also serves as the event type of the event. Key bindings describe events by their types; thus, if there is a key binding for `mouse-1`, that binding would apply to all events whose *event-type* is `mouse-1`.

*position* This is a *mouse position list* specifying where the mouse click occurred; see below for details.

*click-count*

This is the number of rapid repeated presses so far of the same mouse button. See Section 2.7.7 [Repeat Events], page 29.

To access the contents of a mouse position list in the *position* slot of a click event, you should typically use the functions documented in Section 2.7.13 [Accessing Mouse], page 35. The explicit format of the list depends on where the click occurred. For clicks in the text area, mode line, header line, or in the fringe or marginal areas, the mouse position list has the form

```
(window pos-or-area (x . y) timestamp
 object text-pos (col . row)
 image (dx . dy) (width . height))
```

The meanings of these list elements are as follows:

*window* The window in which the click occurred.

*pos-or-area*

The buffer position of the character clicked on in the text area; or, if the click was outside the text area, the window area where it occurred. It is one of the symbols `mode-line`, `header-line`, `vertical-line`, `left-margin`, `right-margin`, `left-fringe`, or `right-fringe`.

In one special case, *pos-or-area* is a list containing a symbol (one of the symbols listed above) instead of just the symbol. This happens after the imaginary prefix keys for the event are registered by Emacs. See Section 2.8.1 [Key Sequence Input], page 39.

*x*, *y* The relative pixel coordinates of the click. For clicks in the text area of a window, the coordinate origin `(0 . 0)` is taken to be the top left corner of the

text area. See Section 17.3 [Window Sizes], page 293. For clicks in a mode line or header line, the coordinate origin is the top left corner of the window itself. For fringes, margins, and the vertical border, *x* does not have meaningful data. For fringes and margins, *y* is relative to the bottom edge of the header line. In all cases, the *x* and *y* coordinates increase rightward and downward respectively.

*timestamp*
: The time at which the event occurred, as an integer number of milliseconds since a system-dependent initial time.

*object*
: Either `nil` if there is no string-type text property at the click position, or a cons cell of the form (*string* . *string-pos*) if there is one:

  *string* The string which was clicked on, including any properties.

  *string-pos* The position in the string where the click occurred.

*text-pos*
: For clicks on a marginal area or on a fringe, this is the buffer position of the first visible character in the corresponding line in the window. For other events, it is the current buffer position in the window.

*col*, *row*
: These are the actual column and row coordinate numbers of the glyph under the *x*, *y* position. If *x* lies beyond the last column of actual text on its line, *col* is reported by adding fictional extra columns that have the default character width. Row 0 is taken to be the header line if the window has one, or the topmost row of the text area otherwise. Column 0 is taken to be the leftmost column of the text area for clicks on a window text area, or the leftmost mode line or header line column for clicks there. For clicks on fringes or vertical borders, these have no meaningful data. For clicks on margins, *col* is measured from the left edge of the margin area and *row* is measured from the top of the margin area.

*image*
: This is the image object on which the click occurred. It is either `nil` if there is no image at the position clicked on, or it is an image object as returned by `find-image` if click was in an image.

*dx*, *dy*
: These are the pixel coordinates of the click, relative to the top left corner of *object*, which is `(0 . 0)`. If *object* is `nil`, the coordinates are relative to the top left corner of the character glyph clicked on.

*width*, *height*
: These are the pixel width and height of *object* or, if this is `nil`, those of the character glyph clicked on.

For clicks on a scroll bar, *position* has this form:

```
(window area (portion . whole) timestamp part)
```

*window*
: The window whose scroll bar was clicked on.

*area*
: This is the symbol `vertical-scroll-bar`.

*portion*
: The number of pixels from the top of the scroll bar to the click position. On some toolkits, including GTK+, Emacs cannot extract this data, so the value is always `0`.

*whole* The total length, in pixels, of the scroll bar. On some toolkits, including GTK+, Emacs cannot extract this data, so the value is always `0`.

*timestamp*
The time at which the event occurred, in milliseconds. On some toolkits, including GTK+, Emacs cannot extract this data, so the value is always `0`.

*part* The part of the scroll bar on which the click occurred. It is one of the symbols `handle` (the scroll bar handle), `above-handle` (the area above the handle), `below-handle` (the area below the handle), `up` (the up arrow at one end of the scroll bar), or `down` (the down arrow at one end of the scroll bar).

### 2.7.5 Drag Events

With Emacs, you can have a drag event without even changing your clothes. A *drag event* happens every time the user presses a mouse button and then moves the mouse to a different character position before releasing the button. Like all mouse events, drag events are represented in Lisp as lists. The lists record both the starting mouse position and the final position, like this:

```
(event-type
 (window1 START-POSITION)
 (window2 END-POSITION))
```

For a drag event, the name of the symbol *event-type* contains the prefix '`drag-`'. For example, dragging the mouse with button 2 held down generates a `drag-mouse-2` event. The second and third elements of the event give the starting and ending position of the drag, as mouse position lists (see Section 2.7.4 [Click Events], page 26). You can access the second element of any mouse event in the same way, with no need to distinguish drag events from others.

The '`drag-`' prefix follows the modifier key prefixes such as '`C-`' and '`M-`'.

If `read-key-sequence` receives a drag event that has no key binding, and the corresponding click event does have a binding, it changes the drag event into a click event at the drag's starting position. This means that you don't have to distinguish between click and drag events unless you want to.

### 2.7.6 Button-Down Events

Click and drag events happen when the user releases a mouse button. They cannot happen earlier, because there is no way to distinguish a click from a drag until the button is released.

If you want to take action as soon as a button is pressed, you need to handle *button-down* events.[2] These occur as soon as a button is pressed. They are represented by lists that look exactly like click events (see Section 2.7.4 [Click Events], page 26), except that the *event-type* symbol name contains the prefix '`down-`'. The '`down-`' prefix follows modifier key prefixes such as '`C-`' and '`M-`'.

The function `read-key-sequence` ignores any button-down events that don't have command bindings; therefore, the Emacs command loop ignores them too. This means that you need not worry about defining button-down events unless you want them to do something. The usual reason to define a button-down event is so that you can track mouse motion

[2] Button-down is the conservative antithesis of drag.

(by reading motion events) until the button is released. See Section 2.7.8 [Motion Events], page 30.

### 2.7.7 Repeat Events

If you press the same mouse button more than once in quick succession without moving the mouse, Emacs generates special *repeat* mouse events for the second and subsequent presses.

The most common repeat events are *double-click* events. Emacs generates a double-click event when you click a button twice; the event happens when you release the button (as is normal for all click events).

The event type of a double-click event contains the prefix '`double-`'. Thus, a double click on the second mouse button with META held down comes to the Lisp program as `M-double-mouse-2`. If a double-click event has no binding, the binding of the corresponding ordinary click event is used to execute it. Thus, you need not pay attention to the double click feature unless you really want to.

When the user performs a double click, Emacs generates first an ordinary click event, and then a double-click event. Therefore, you must design the command binding of the double click event to assume that the single-click command has already run. It must produce the desired results of a double click, starting from the results of a single click.

This is convenient, if the meaning of a double click somehow "builds on" the meaning of a single click—which is recommended user interface design practice for double clicks.

If you click a button, then press it down again and start moving the mouse with the button held down, then you get a *double-drag* event when you ultimately release the button. Its event type contains '`double-drag`' instead of just '`drag`'. If a double-drag event has no binding, Emacs looks for an alternate binding as if the event were an ordinary drag.

Before the double-click or double-drag event, Emacs generates a *double-down* event when the user presses the button down for the second time. Its event type contains '`double-down`' instead of just '`down`'. If a double-down event has no binding, Emacs looks for an alternate binding as if the event were an ordinary button-down event. If it finds no binding that way either, the double-down event is ignored.

To summarize, when you click a button and then press it again right away, Emacs generates a down event and a click event for the first click, a double-down event when you press the button again, and finally either a double-click or a double-drag event.

If you click a button twice and then press it again, all in quick succession, Emacs generates a *triple-down* event, followed by either a *triple-click* or a *triple-drag*. The event types of these events contain '`triple`' instead of '`double`'. If any triple event has no binding, Emacs uses the binding that it would use for the corresponding double event.

If you click a button three or more times and then press it again, the events for the presses beyond the third are all triple events. Emacs does not have separate event types for quadruple, quintuple, etc. events. However, you can look at the event list to find out precisely how many times the button was pressed.

`event-click-count` *event* [Function]

This function returns the number of consecutive button presses that led up to *event*. If *event* is a double-down, double-click or double-drag event, the value is 2. If *event* is a triple event, the value is 3 or greater. If *event* is an ordinary mouse event (not a repeat event), the value is 1.

`double-click-fuzz` [User Option]

To generate repeat events, successive mouse button presses must be at approximately the same screen position. The value of `double-click-fuzz` specifies the maximum number of pixels the mouse may be moved (horizontally or vertically) between two successive clicks to make a double-click.

This variable is also the threshold for motion of the mouse to count as a drag.

`double-click-time` [User Option]

To generate repeat events, the number of milliseconds between successive button presses must be less than the value of `double-click-time`. Setting `double-click-time` to `nil` disables multi-click detection entirely. Setting it to `t` removes the time limit; Emacs then detects multi-clicks by position only.

### 2.7.8 Motion Events

Emacs sometimes generates *mouse motion* events to describe motion of the mouse without any button activity. Mouse motion events are represented by lists that look like this:

```
(mouse-movement POSITION)
```

*position* is a mouse position list (see Section 2.7.4 [Click Events], page 26), specifying the current position of the mouse cursor.

The special form `track-mouse` enables generation of motion events within its body. Outside of `track-mouse` forms, Emacs does not generate events for mere motion of the mouse, and these events do not appear. See Section 18.13 [Mouse Tracking], page 362.

### 2.7.9 Focus Events

Window systems provide general ways for the user to control which window gets keyboard input. This choice of window is called the *focus*. When the user does something to switch between Emacs frames, that generates a *focus event*. The normal definition of a focus event, in the global keymap, is to select a new frame within Emacs, as the user would expect. See Section 18.9 [Input Focus], page 358.

Focus events are represented in Lisp as lists that look like this:

```
(switch-frame new-frame)
```

where *new-frame* is the frame switched to.

Some X window managers are set up so that just moving the mouse into a window is enough to set the focus there. Usually, there is no need for a Lisp program to know about the focus change until some other kind of input arrives. Emacs generates a focus event only when the user actually types a keyboard key or presses a mouse button in the new frame; just moving the mouse between frames does not generate a focus event.

A focus event in the middle of a key sequence would garble the sequence. So Emacs never generates a focus event in the middle of a key sequence. If the user changes focus in the middle of a key sequence—that is, after a prefix key—then Emacs reorders the events so that the focus event comes either before or after the multi-event key sequence, and not within it.

### 2.7.10 Miscellaneous System Events

A few other event types represent occurrences within the system.

`(delete-frame (`*frame*`))`
: This kind of event indicates that the user gave the window manager a command to delete a particular window, which happens to be an Emacs frame.

  The standard definition of the `delete-frame` event is to delete *frame*.

`(iconify-frame (`*frame*`))`
: This kind of event indicates that the user iconified *frame* using the window manager. Its standard definition is `ignore`; since the frame has already been iconified, Emacs has no work to do. The purpose of this event type is so that you can keep track of such events if you want to.

`(make-frame-visible (`*frame*`))`
: This kind of event indicates that the user deiconified *frame* using the window manager. Its standard definition is `ignore`; since the frame has already been made visible, Emacs has no work to do.

`(wheel-up` *position*`)`
`(wheel-down` *position*`)`
: These kinds of event are generated by moving a mouse wheel. The *position* element is a mouse position list (see Section 2.7.4 [Click Events], page 26), specifying the position of the mouse cursor when the event occurred.

  This kind of event is generated only on some kinds of systems. On some systems, `mouse-4` and `mouse-5` are used instead. For portable code, use the variables `mouse-wheel-up-event` and `mouse-wheel-down-event` defined in '`mwheel.el`' to determine what event types to expect for the mouse wheel.

`(drag-n-drop` *position files*`)`
: This kind of event is generated when a group of files is selected in an application outside of Emacs, and then dragged and dropped onto an Emacs frame.

  The element *position* is a list describing the position of the event, in the same format as used in a mouse-click event (see Section 2.7.4 [Click Events], page 26), and *files* is the list of file names that were dragged and dropped. The usual way to handle this event is by visiting these files.

  This kind of event is generated, at present, only on some kinds of systems.

`help-echo`
: This kind of event is generated when a mouse pointer moves onto a portion of buffer text which has a `help-echo` text property. The generated event has this form:

  ```
  (help-echo frame help window object pos)
  ```

  The precise meaning of the event parameters and the way these parameters are used to display the help-echo text are described in [Text help-echo], page 495.

`sigusr1`
`sigusr2`
: These events are generated when the Emacs process receives the signals `SIGUSR1` and `SIGUSR2`. They contain no additional data because signals do not carry

additional information. They can be useful for debugging (see ⟨undefined⟩ [Error Debugging], page ⟨undefined⟩).

To catch a user signal, bind the corresponding event to an interactive command in the `special-event-map` (see ⟨undefined⟩ [Active Keymaps], page ⟨undefined⟩). The command is called with no arguments, and the specific signal event is available in `last-input-event`. For example:

```
(defun sigusr-handler ()
  (interactive)
  (message "Caught signal %S" last-input-event))

(define-key special-event-map [sigusr1] 'sigusr-handler)
```

To test the signal handler, you can make Emacs send a signal to itself:

```
(signal-process (emacs-pid) 'sigusr1)
```

`language-change`

This kind of event is generated on MS-Windows when the input language has changed. This typically means that the keyboard keys will send to Emacs characters from a different language. The generated event has this form:

```
(language-change frame codepage language-id)
```

Here *frame* is the frame which was current when the input language changed; *codepage* is the new codepage number; and *language-id* is the numerical ID of the new input language. The coding-system (see Section 19.6 [Coding Systems], page 381) that corresponds to *codepage* is `cp`*codepage* or `windows-`*codepage*. To convert *language-id* to a string (e.g., to use it for various language-dependent features, such as `set-language-environment`), use the `w32-get-locale-info` function, like this:

```
;; Get the abbreviated language name, such as "ENU" for English
(w32-get-locale-info language-id)
;; Get the full English name of the language,
;; such as "English (United States)"
(w32-get-locale-info language-id 4097)
;; Get the full localized name of the language
(w32-get-locale-info language-id t)
```

If one of these events arrives in the middle of a key sequence—that is, after a prefix key—then Emacs reorders the events so that this event comes either before or after the multi-event key sequence, not within it.

### 2.7.11 Event Examples

If the user presses and releases the left mouse button over the same location, that generates a sequence of events like this:

```
(down-mouse-1 (#<window 18 on NEWS> 2613 (0 . 38) -864320))
(mouse-1      (#<window 18 on NEWS> 2613 (0 . 38) -864180))
```

While holding the control key down, the user might hold down the second mouse button, and drag the mouse from one line to the next. That produces two events, as shown here:

```
(C-down-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219))
(C-drag-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219)
                (#<window 18 on NEWS> 3510 (0 . 28) -729648))
```

While holding down the meta and shift keys, the user might press the second mouse button on the window's mode line, and then drag the mouse into another window. That produces a pair of events like these:

```
(M-S-down-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844))
(M-S-drag-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844)
                  (#<window 20 on carlton-sanskrit.tex> 161 (33 . 3)
                   -453816))
```

To handle a SIGUSR1 signal, define an interactive function, and bind it to the `signal usr1` event sequence:

```
(defun usr1-handler ()
  (interactive)
  (message "Got USR1 signal"))
(global-set-key [signal usr1] 'usr1-handler)
```

### 2.7.12 Classifying Events

Every event has an *event type*, which classifies the event for key binding purposes. For a keyboard event, the event type equals the event value; thus, the event type for a character is the character, and the event type for a function key symbol is the symbol itself. For events that are lists, the event type is the symbol in the CAR of the list. Thus, the event type is always a symbol or a character.

Two events of the same type are equivalent where key bindings are concerned; thus, they always run the same command. That does not necessarily mean they do the same things, however, as some commands look at the whole event to decide what to do. For example, some commands use the location of a mouse event to decide where in the buffer to act.

Sometimes broader classifications of events are useful. For example, you might want to ask whether an event involved the META key, regardless of which other key or mouse button was used.

The functions `event-modifiers` and `event-basic-type` are provided to get such information conveniently.

`event-modifiers` *event* [Function]

This function returns a list of the modifiers that *event* has. The modifiers are symbols; they include `shift`, `control`, `meta`, `alt`, `hyper` and `super`. In addition, the modifiers list of a mouse event symbol always contains one of `click`, `drag`, and `down`. For double or triple events, it also contains `double` or `triple`.

The argument *event* may be an entire event object, or just an event type. If *event* is a symbol that has never been used in an event that has been read as input in the current Emacs session, then `event-modifiers` can return `nil`, even when *event* actually has modifiers.

Here are some examples:

```
(event-modifiers ?a)
     ⇒ nil
(event-modifiers ?A)
     ⇒ (shift)
(event-modifiers ?\C-a)
     ⇒ (control)
(event-modifiers ?\C-%)
```

```
     ⇒ (control)
(event-modifiers ?\C-\S-a)
     ⇒ (control shift)
(event-modifiers 'f5)
     ⇒ nil
(event-modifiers 's-f5)
     ⇒ (super)
(event-modifiers 'M-S-f5)
     ⇒ (meta shift)
(event-modifiers 'mouse-1)
     ⇒ (click)
(event-modifiers 'down-mouse-1)
     ⇒ (down)
```

The modifiers list for a click event explicitly contains `click`, but the event symbol name itself does not contain '`click`'.

`event-basic-type` *event* [Function]

This function returns the key or mouse button that *event* describes, with all modifiers removed. The *event* argument is as in `event-modifiers`. For example:

```
(event-basic-type ?a)
     ⇒ 97
(event-basic-type ?A)
     ⇒ 97
(event-basic-type ?\C-a)
     ⇒ 97
(event-basic-type ?\C-\S-a)
     ⇒ 97
(event-basic-type 'f5)
     ⇒ f5
(event-basic-type 's-f5)
     ⇒ f5
(event-basic-type 'M-S-f5)
     ⇒ f5
(event-basic-type 'down-mouse-1)
     ⇒ mouse-1
```

`mouse-movement-p` *object* [Function]

This function returns non-`nil` if *object* is a mouse movement event.

`event-convert-list` *list* [Function]

This function converts a list of modifier names and a basic event type to an event type which specifies all of them. The basic event type must be the last element of the list. For example,

```
(event-convert-list '(control ?a))
     ⇒ 1
(event-convert-list '(control meta ?a))
     ⇒ -134217727
```

```
(event-convert-list '(control super f1))
     ⇒ C-s-f1
```

### 2.7.13 Accessing Mouse Events

This section describes convenient functions for accessing the data in a mouse button or motion event.

The following two functions return a mouse position list (see Section 2.7.4 [Click Events], page 26), specifying the position of a mouse event.

`event-start` *event* [Function]

This returns the starting position of *event*.

If *event* is a click or button-down event, this returns the location of the event. If *event* is a drag event, this returns the drag's starting position.

`event-end` *event* [Function]

This returns the ending position of *event*.

If *event* is a drag event, this returns the position where the user released the mouse button. If *event* is a click or button-down event, the value is actually the starting position, which is the only position such events have.

`posnp` *object* [Function]

This function returns non-`nil` if *object* is a mouse position list, in either of the formats documented in Section 2.7.4 [Click Events], page 26); and `nil` otherwise.

These functions take a mouse position list as argument, and return various parts of it:

`posn-window` *position* [Function]

Return the window that *position* is in.

`posn-area` *position* [Function]

Return the window area recorded in *position*. It returns `nil` when the event occurred in the text area of the window; otherwise, it is a symbol identifying the area in which the event occurred.

`posn-point` *position* [Function]

Return the buffer position in *position*. When the event occurred in the text area of the window, in a marginal area, or on a fringe, this is an integer specifying a buffer position. Otherwise, the value is undefined.

`posn-x-y` *position* [Function]

Return the pixel-based x and y coordinates in *position*, as a cons cell `(x . y)`. These coordinates are relative to the window given by `posn-window`.

This example shows how to convert the window-relative coordinates in the text area of a window into frame-relative coordinates:

```
(defun frame-relative-coordinates (position)
  "Return frame-relative coordinates from POSITION.
POSITION is assumed to lie in a window text area."
  (let* ((x-y (posn-x-y position))
```

```
      (window (posn-window position))
      (edges (window-inside-pixel-edges window)))
 (cons (+ (car x-y) (car edges))
       (+ (cdr x-y) (cadr edges)))))
```

`posn-col-row` *position* [Function]
: This function returns a cons cell (*col* . *row*), containing the estimated column and row corresponding to buffer position *position*. The return value is given in units of the frame's default character width and height, as computed from the *x* and *y* values corresponding to *position*. (So, if the actual characters have non-default sizes, the actual row and column may differ from these computed values.)

  Note that *row* is counted from the top of the text area. If the window possesses a header line (see Section 20.4.7 [Header Lines], page 426), it is *not* counted as the first line.

`posn-actual-col-row` *position* [Function]
: Return the actual row and column in *position*, as a cons cell (*col* . *row*). The values are the actual row and column numbers in the window. See Section 2.7.4 [Click Events], page 26, for details. It returns `nil` if *position* does not include actual positions values.

`posn-string` *position* [Function]
: Return the string object in *position*, either `nil`, or a cons cell (*string* . *string-pos*).

`posn-image` *position* [Function]
: Return the image object in *position*, either `nil`, or an image (`image ...`).

`posn-object` *position* [Function]
: Return the image or string object in *position*, either `nil`, an image (`image ...`), or a cons cell (*string* . *string-pos*).

`posn-object-x-y` *position* [Function]
: Return the pixel-based x and y coordinates relative to the upper left corner of the object in *position* as a cons cell (*dx* . *dy*). If the *position* is a buffer position, return the relative position in the character at that position.

`posn-object-width-height` *position* [Function]
: Return the pixel width and height of the object in *position* as a cons cell (*width* . *height*). If the *position* is a buffer position, return the size of the character at that position.

`posn-timestamp` *position* [Function]
: Return the timestamp in *position*. This is the time at which the event occurred, in milliseconds.

These functions compute a position list given particular buffer position or screen position. You can access the data in this position list with the functions described above.

`posn-at-point` **&optional** *pos window* [Function]
: This function returns a position list for position *pos* in *window*. *pos* defaults to point in *window*; *window* defaults to the selected window.

  `posn-at-point` returns `nil` if *pos* is not visible in *window*.

`posn-at-x-y` *x y* **&optional** *frame-or-window whole* [Function]
: This function returns position information corresponding to pixel coordinates *x* and *y* in a specified frame or window, *frame-or-window*, which defaults to the selected window. The coordinates *x* and *y* are relative to the frame or window used. If *whole* is `nil`, the coordinates are relative to the window text area, otherwise they are relative to the entire window area including scroll bars, margins and fringes.

### 2.7.14 Accessing Scroll Bar Events

These functions are useful for decoding scroll bar events.

`scroll-bar-event-ratio` *event* [Function]
: This function returns the fractional vertical position of a scroll bar event within the scroll bar. The value is a cons cell (`portion . whole`) containing two integers whose ratio is the fractional position.

`scroll-bar-scale` *ratio total* [Function]
: This function multiplies (in effect) *ratio* by *total*, rounding the result to an integer. The argument *ratio* is not a number, but rather a pair (`num . denom`)—typically a value returned by `scroll-bar-event-ratio`.

  This function is handy for scaling a position on a scroll bar into a buffer position. Here's how to do that:

```
(+ (point-min)
   (scroll-bar-scale
      (posn-x-y (event-start event))
      (- (point-max) (point-min))))
```

  Recall that scroll bar events have two integers forming a ratio, in place of a pair of x and y coordinates.

### 2.7.15 Putting Keyboard Events in Strings

In most of the places where strings are used, we conceptualize the string as containing text characters—the same kind of characters found in buffers or files. Occasionally Lisp programs use strings that conceptually contain keyboard characters; for example, they may be key sequences or keyboard macro definitions. However, storing keyboard characters in a string is a complex matter, for reasons of historical compatibility, and it is not always possible.

We recommend that new programs avoid dealing with these complexities by not storing keyboard events in strings. Here is how to do that:

- Use vectors instead of strings for key sequences, when you plan to use them for anything other than as arguments to `lookup-key` and `define-key`. For example, you can use `read-key-sequence-vector` instead of `read-key-sequence`, and `this-command-keys-vector` instead of `this-command-keys`.

- Use vectors to write key sequence constants containing meta characters, even when passing them directly to `define-key`.
- When you have to look at the contents of a key sequence that might be a string, use `listify-key-sequence` (see Section 2.8.6 [Event Input Misc], page 45) first, to convert it to a list.

The complexities stem from the modifier bits that keyboard input characters can include. Aside from the Meta modifier, none of these modifier bits can be included in a string, and the Meta modifier is allowed only in special cases.

The earliest GNU Emacs versions represented meta characters as codes in the range of 128 to 255. At that time, the basic character codes ranged from 0 to 127, so all keyboard character codes did fit in a string. Many Lisp programs used '`\M-`' in string constants to stand for meta characters, especially in arguments to `define-key` and similar functions, and key sequences and sequences of events were always represented as strings.

When we added support for larger basic character codes beyond 127, and additional modifier bits, we had to change the representation of meta characters. Now the flag that represents the Meta modifier in a character is $2^{27}$ and such numbers cannot be included in a string.

To support programs with '`\M-`' in string constants, there are special rules for including certain meta characters in a string. Here are the rules for interpreting a string as a sequence of input characters:

- If the keyboard character value is in the range of 0 to 127, it can go in the string unchanged.
- The meta variants of those characters, with codes in the range of $2^{27}$ to $2^{27} + 127$, can also go in the string, but you must change their numeric values. You must set the $2^7$ bit instead of the $2^{27}$ bit, resulting in a value between 128 and 255. Only a unibyte string can include these codes.
- Non-ASCII characters above 256 can be included in a multibyte string.
- Other keyboard character events cannot fit in a string. This includes keyboard events in the range of 128 to 255.

Functions such as `read-key-sequence` that construct strings of keyboard input characters follow these rules: they construct vectors instead of strings, when the events won't fit in a string.

When you use the read syntax '`\M-`' in a string, it produces a code in the range of 128 to 255—the same code that you get if you modify the corresponding keyboard event to put it in the string. Thus, meta events in strings work consistently regardless of how they get into the strings.

However, most programs would do well to avoid these issues by following the recommendations at the beginning of this section.

## 2.8 Reading Input

The editor command loop reads key sequences using the function `read-key-sequence`, which uses `read-event`. These and other functions for event input are also available for use in Lisp programs. See also `momentary-string-display` in Section 11.8 [Temporary

Displays], page 125, and `sit-for` in Section 2.10 [Waiting], page 47. See ⟨undefined⟩ [Terminal Input], page ⟨undefined⟩, for functions and variables for controlling terminal input modes and debugging terminal input.

For higher-level input facilities, see ⟨undefined⟩ [Minibuffers], page ⟨undefined⟩.

### 2.8.1 Key Sequence Input

The command loop reads input a key sequence at a time, by calling `read-key-sequence`. Lisp programs can also call this function; for example, `describe-key` uses it to read the key to describe.

`read-key-sequence` *prompt* **&optional** *continue-echo dont-downcase-last switch-frame-ok command-loop* [Function]

This function reads a key sequence and returns it as a string or vector. It keeps reading events until it has accumulated a complete key sequence; that is, enough to specify a non-prefix command using the currently active keymaps. (Remember that a key sequence that starts with a mouse event is read using the keymaps of the buffer in the window that the mouse was in, not the current buffer.)

If the events are all characters and all can fit in a string, then `read-key-sequence` returns a string (see Section 2.7.15 [Strings of Events], page 37). Otherwise, it returns a vector, since a vector can hold all kinds of events—characters, symbols, and lists. The elements of the string or vector are the events in the key sequence.

Reading a key sequence includes translating the events in various ways. See ⟨undefined⟩ [Translation Keymaps], page ⟨undefined⟩.

The argument *prompt* is either a string to be displayed in the echo area as a prompt, or `nil`, meaning not to display a prompt. The argument *continue-echo*, if non-`nil`, means to echo this key as a continuation of the previous key.

Normally any upper case event is converted to lower case if the original event is undefined and the lower case equivalent is defined. The argument *dont-downcase-last*, if non-`nil`, means do not convert the last event to lower case. This is appropriate for reading a key sequence to be defined.

The argument *switch-frame-ok*, if non-`nil`, means that this function should process a `switch-frame` event if the user switches frames before typing anything. If the user switches frames in the middle of a key sequence, or at the start of the sequence but *switch-frame-ok* is `nil`, then the event will be put off until after the current key sequence.

The argument *command-loop*, if non-`nil`, means that this key sequence is being read by something that will read commands one after another. It should be `nil` if the caller will read just one key sequence.

In the following example, Emacs displays the prompt '`?`' in the echo area, and then the user types `C-x C-f`.

```
(read-key-sequence "?")
```

```
---------- Echo Area ----------
?C-x C-f
---------- Echo Area ----------

     ⇒ "^X^F"
```

The function `read-key-sequence` suppresses quitting: `C-g` typed while reading with this function works like any other character, and does not set `quit-flag`. See Section 34.1 [Quitting], page 717.

`read-key-sequence-vector` *prompt* **&optional** *continue-echo dont-downcase-last switch-frame-ok command-loop* [Function]

This is like `read-key-sequence` except that it always returns the key sequence as a vector, never as a string. See Section 2.7.15 [Strings of Events], page 37.

If an input character is upper-case (or has the shift modifier) and has no key binding, but its lower-case equivalent has one, then `read-key-sequence` converts the character to lower case. Note that `lookup-key` does not perform case conversion in this way.

When reading input results in such a *shift-translation*, Emacs sets the variable `this-command-keys-shift-translated` to a non-`nil` value. Lisp programs can examine this variable if they need to modify their behavior when invoked by shift-translated keys. For example, the function `handle-shift-selection` examines the value of this variable to determine how to activate or deactivate the region (see ⟨undefined⟩ [The Mark], page ⟨undefined⟩).

The function `read-key-sequence` also transforms some mouse events. It converts unbound drag events into click events, and discards unbound button-down events entirely. It also reshuffles focus events and miscellaneous window events so that they never appear in a key sequence with any other events.

When mouse events occur in special parts of a window, such as a mode line or a scroll bar, the event type shows nothing special—it is the same symbol that would normally represent that combination of mouse button and modifier keys. The information about the window part is kept elsewhere in the event—in the coordinates. But `read-key-sequence` translates this information into imaginary "prefix keys", all of which are symbols: `header-line`, `horizontal-scroll-bar`, `menu-bar`, `mode-line`, `vertical-line`, and `vertical-scroll-bar`. You can define meanings for mouse clicks in special window parts by defining key sequences using these imaginary prefix keys.

For example, if you call `read-key-sequence` and then click the mouse on the window's mode line, you get two events, like this:

```
(read-key-sequence "Click on the mode line: ")
     ⇒ [mode-line
        (mouse-1
         (#<window 6 on NEWS> mode-line
          (40 . 63) 5959987))]
```

`num-input-keys` [Variable]

This variable's value is the number of key sequences processed so far in this Emacs session. This includes key sequences read from the terminal and key sequences read from keyboard macros being executed.

### 2.8.2 Reading One Event

The lowest level functions for command input are `read-event`, `read-char`, and `read-char-exclusive`.

`read-event` **&optional** *prompt inherit-input-method seconds* [Function]

This function reads and returns the next event of command input, waiting if necessary until an event is available. Events can come directly from the user or from a keyboard macro.

If the optional argument *prompt* is non-`nil`, it should be a string to display in the echo area as a prompt. Otherwise, `read-event` does not display any message to indicate it is waiting for input; instead, it prompts by echoing: it displays descriptions of the events that led to or were read by the current command. See Section 11.4 [The Echo Area], page 114.

If *inherit-input-method* is non-`nil`, then the current input method (if any) is employed to make it possible to enter a non-ASCII character. Otherwise, input method handling is disabled for reading this event.

If `cursor-in-echo-area` is non-`nil`, then `read-event` moves the cursor temporarily to the echo area, to the end of any message displayed there. Otherwise `read-event` does not move the cursor.

If *seconds* is non-`nil`, it should be a number specifying the maximum time to wait for input, in seconds. If no input arrives within that time, `read-event` stops waiting and returns `nil`. A floating-point value for *seconds* means to wait for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down. If *seconds* is `nil`, `read-event` waits as long as necessary for input to arrive.

If *seconds* is `nil`, Emacs is considered idle while waiting for user input to arrive. Idle timers—those created with `run-with-idle-timer` (see ⟨undefined⟩ [Idle Timers], page ⟨undefined⟩)—can run during this period. However, if *seconds* is non-`nil`, the state of idleness remains unchanged. If Emacs is non-idle when `read-event` is called, it remains non-idle throughout the operation of `read-event`; if Emacs is idle (which can happen if the call happens inside an idle timer), it remains idle.

If `read-event` gets an event that is defined as a help character, then in some cases `read-event` processes the event directly without returning. See Section 7.5 [Help Functions], page 85. Certain other events, called *special events*, are also processed directly within `read-event` (see Section 2.9 [Special Events], page 46).

Here is what happens if you call `read-event` and then press the right-arrow function key:

```
(read-event)
     ⇒ right
```

`read-char` **&optional** *prompt inherit-input-method seconds* [Function]

This function reads and returns a character of command input. If the user generates an event which is not a character (i.e., a mouse click or function key event), `read-char` signals an error. The arguments work as in `read-event`.

In the first example, the user types the character `1` (ASCII code 49). The second example shows a keyboard macro definition that calls `read-char` from the minibuffer

using `eval-expression`. `read-char` reads the keyboard macro's very next character, which is `1`. Then `eval-expression` displays its return value in the echo area.

```
(read-char)
     ⇒ 49

;; We assume here you use M-: to evaluate this.
(symbol-function 'foo)
     ⇒ "^[:(read-char)^M1"
(execute-kbd-macro 'foo)
     ⊣ 49
     ⇒ nil
```

`read-char-exclusive` **&optional** *prompt inherit-input-method seconds* [Function]
: This function reads and returns a character of command input. If the user generates an event which is not a character, `read-char-exclusive` ignores it and reads another event, until it gets a character. The arguments work as in `read-event`.

None of the above functions suppress quitting.

`num-nonmacro-input-events` [Variable]
: This variable holds the total number of input events received so far from the terminal—not counting those generated by keyboard macros.

We emphasize that, unlike `read-key-sequence`, the functions `read-event`, `read-char`, and `read-char-exclusive` do not perform the translations described in ⟨undefined⟩ [Translation Keymaps], page ⟨undefined⟩. If you wish to read a single key taking these translations into account, use the function `read-key`:

`read-key` **&optional** *prompt* [Function]
: This function reads a single key. It is "intermediate" between `read-key-sequence` and `read-event`. Unlike the former, it reads a single key, not a key sequence. Unlike the latter, it does not return a raw event, but decodes and translates the user input according to `input-decode-map`, `local-function-key-map`, and `key-translation-map` (see ⟨undefined⟩ [Translation Keymaps], page ⟨undefined⟩).

  The argument *prompt* is either a string to be displayed in the echo area as a prompt, or `nil`, meaning not to display a prompt.

`read-char-choice` *prompt chars* **&optional** *inhibit-quit* [Function]
: This function uses `read-key` to read and return a single character. It ignores any input that is not a member of *chars*, a list of accepted characters. Optionally, it will also ignore keyboard-quit events while it is waiting for valid input. If you bind `help-form` (see Section 7.5 [Help Functions], page 85) to a non-`nil` value while calling `read-char-choice`, then pressing `help-char` causes it to evaluate `help-form` and display the result. It then continues to wait for a valid input character, or keyboard-quit.

### 2.8.3 Modifying and Translating Input Events

Emacs modifies every event it reads according to `extra-keyboard-modifiers`, then translates it through `keyboard-translate-table` (if applicable), before returning it from `read-event`.

`extra-keyboard-modifiers` [Variable]

This variable lets Lisp programs "press" the modifier keys on the keyboard. The value is a character. Only the modifiers of the character matter. Each time the user types a keyboard key, it is altered as if those modifier keys were held down. For instance, if you bind `extra-keyboard-modifiers` to `?\C-\M-a`, then all keyboard input characters typed during the scope of the binding will have the control and meta modifiers applied to them. The character `?\C-@`, equivalent to the integer 0, does not count as a control character for this purpose, but as a character with no modifiers. Thus, setting `extra-keyboard-modifiers` to zero cancels any modification.

When using a window system, the program can "press" any of the modifier keys in this way. Otherwise, only the CTL and META keys can be virtually pressed.

Note that this variable applies only to events that really come from the keyboard, and has no effect on mouse events or any other events.

`keyboard-translate-table` [Variable]

This terminal-local variable is the translate table for keyboard characters. It lets you reshuffle the keys on the keyboard without changing any command bindings. Its value is normally a char-table, or else `nil`. (It can also be a string or vector, but this is considered obsolete.)

If `keyboard-translate-table` is a char-table (see ⟨undefined⟩ [Char-Tables], page ⟨undefined⟩), then each character read from the keyboard is looked up in this char-table. If the value found there is non-`nil`, then it is used instead of the actual input character.

Note that this translation is the first thing that happens to a character after it is read from the terminal. Record-keeping features such as `recent-keys` and dribble files record the characters after translation.

Note also that this translation is done before the characters are supplied to input methods (see Section 19.4 [Input Methods], page 378). Use `translation-table-for-input` (see ⟨undefined⟩ [Translation of Characters], page ⟨undefined⟩), if you want to translate characters after input methods operate.

`keyboard-translate` *from to* [Function]

This function modifies `keyboard-translate-table` to translate character code *from* into character code *to*. It creates the keyboard translate table if necessary.

Here's an example of using the `keyboard-translate-table` to make `C-x`, `C-c` and `C-v` perform the cut, copy and paste operations:

```
(keyboard-translate ?\C-x 'control-x)
(keyboard-translate ?\C-c 'control-c)
(keyboard-translate ?\C-v 'control-v)
(global-set-key [control-x] 'kill-region)
(global-set-key [control-c] 'kill-ring-save)
(global-set-key [control-v] 'yank)
```

On a graphical terminal that supports extended ASCII input, you can still get the standard Emacs meanings of one of those characters by typing it with the shift key. That makes it a different character as far as keyboard translation is concerned, but it has the same usual meaning.

See ⟨undefined⟩ [Translation Keymaps], page ⟨undefined⟩, for mechanisms that translate event sequences at the level of `read-key-sequence`.

### 2.8.4 Invoking the Input Method

The event-reading functions invoke the current input method, if any (see Section 19.4 [Input Methods], page 378). If the value of `input-method-function` is non-`nil`, it should be a function; when `read-event` reads a printing character (including SPC) with no modifier bits, it calls that function, passing the character as an argument.

`input-method-function` [Variable]

If this is non-`nil`, its value specifies the current input method function.

**Warning:** don't bind this variable with `let`. It is often buffer-local, and if you bind it around reading input (which is exactly when you *would* bind it), switching buffers asynchronously while Emacs is waiting will cause the value to be restored in the wrong buffer.

The input method function should return a list of events which should be used as input. (If the list is `nil`, that means there is no input, so `read-event` waits for another event.) These events are processed before the events in `unread-command-events` (see Section 2.8.6 [Event Input Misc], page 45). Events returned by the input method function are not passed to the input method function again, even if they are printing characters with no modifier bits.

If the input method function calls `read-event` or `read-key-sequence`, it should bind `input-method-function` to `nil` first, to prevent recursion.

The input method function is not called when reading the second and subsequent events of a key sequence. Thus, these characters are not subject to input method processing. The input method function should test the values of `overriding-local-map` and `overriding-terminal-local-map`; if either of these variables is non-`nil`, the input method should put its argument into a list and return that list with no further processing.

### 2.8.5 Quoted Character Input

You can use the function `read-quoted-char` to ask the user to specify a character, and allow the user to specify a control or meta character conveniently, either literally or as an octal character code. The command `quoted-insert` uses this function.

`read-quoted-char` **&optional** *prompt* [Function]

This function is like `read-char`, except that if the first character read is an octal digit (0–7), it reads any number of octal digits (but stopping if a non-octal digit is found), and returns the character represented by that numeric character code. If the character that terminates the sequence of octal digits is RET, it is discarded. Any other terminating character is used as input after this function returns.

Quitting is suppressed when the first character is read, so that the user can enter a `C-g`. See Section 34.1 [Quitting], page 717.

If *prompt* is supplied, it specifies a string for prompting the user. The prompt string is always displayed in the echo area, followed by a single '`-`'.

In the following example, the user types in the octal number 177 (which is 127 in decimal).

```
(read-quoted-char "What character")

---------- Echo Area ----------
What character 1 7 7-
---------- Echo Area ----------

     ⇒ 127
```

### 2.8.6 Miscellaneous Event Input Features

This section describes how to "peek ahead" at events without using them up, how to check for pending input, and how to discard pending input. See also the function `read-passwd` (see ⟨undefined⟩ [Reading a Password], page ⟨undefined⟩).

`unread-command-events` [Variable]

This variable holds a list of events waiting to be read as command input. The events are used in the order they appear in the list, and removed one by one as they are used.

The variable is needed because in some cases a function reads an event and then decides not to use it. Storing the event in this variable causes it to be processed normally, by the command loop or by the functions to read command input.

For example, the function that implements numeric prefix arguments reads any number of digits. When it finds a non-digit event, it must unread the event so that it can be read normally by the command loop. Likewise, incremental search uses this feature to unread events with no special meaning in a search, because these events should exit the search and then execute normally.

The reliable and easy way to extract events from a key sequence so as to put them in `unread-command-events` is to use `listify-key-sequence` (see below).

Normally you add events to the front of this list, so that the events most recently unread will be reread first.

Events read from this list are not normally added to the current command's key sequence (as returned by, e.g., `this-command-keys`), as the events will already have been added once as they were read for the first time. An element of the form `(t . event)` forces *event* to be added to the current command's key sequence.

`listify-key-sequence` *key* [Function]

This function converts the string or vector *key* to a list of individual events, which you can put in `unread-command-events`.

`input-pending-p` [Function]

This function determines whether any command input is currently available to be read. It returns immediately, with value `t` if there is available input, `nil` otherwise. On rare occasions it may return `t` when no input is available.

`last-input-event` [Variable]

This variable records the last terminal input event read, whether as part of a command or explicitly by a Lisp program.

In the example below, the Lisp program reads the character `1`, ASCII code 49. It becomes the value of `last-input-event`, while `C-e` (we assume `C-x C-e` command is used to evaluate this expression) remains the value of `last-command-event`.

```
(progn (print (read-char))
       (print last-command-event)
       last-input-event)
 ⊣ 49
 ⊣ 5
⇒ 49
```

`while-no-input` *body...* [Macro]

This construct runs the *body* forms and returns the value of the last one—but only if no input arrives. If any input arrives during the execution of the *body* forms, it aborts them (working much like a quit). The `while-no-input` form returns `nil` if aborted by a real quit, and returns `t` if aborted by arrival of other input.

If a part of *body* binds `inhibit-quit` to non-`nil`, arrival of input during those parts won't cause an abort until the end of that part.

If you want to be able to distinguish all possible values computed by *body* from both kinds of abort conditions, write the code like this:

```
(while-no-input
  (list
    (progn . body)))
```

`discard-input` [Function]

This function discards the contents of the terminal input buffer and cancels any keyboard macro that might be in the process of definition. It returns `nil`.

In the following example, the user may type a number of characters right after starting the evaluation of the form. After the `sleep-for` finishes sleeping, `discard-input` discards any characters typed during the sleep.

```
(progn (sleep-for 2)
       (discard-input))
     ⇒ nil
```

## 2.9 Special Events

Certain *special events* are handled at a very low level—as soon as they are read. The `read-event` function processes these events itself, and never returns them. Instead, it keeps waiting for the first event that is not special and returns that one.

Special events do not echo, they are never grouped into key sequences, and they never appear in the value of `last-command-event` or `(this-command-keys)`. They do not discard a numeric argument, they cannot be unread with `unread-command-events`, they may not appear in a keyboard macro, and they are not recorded in a keyboard macro while you are defining one.

Special events do, however, appear in `last-input-event` immediately after they are read, and this is the way for the event's definition to find the actual event.

The events types `iconify-frame`, `make-frame-visible`, `delete-frame`, `drag-n-drop`, `language-change`, and user signals like `sigusr1` are normally handled in this way. The keymap which defines how to handle special events—and which events are special—is in the variable `special-event-map` (see ⟨undefined⟩ [Active Keymaps], page ⟨undefined⟩).

## 2.10 Waiting for Elapsed Time or Input

The wait functions are designed to wait for a certain amount of time to pass or until there is input. For example, you may wish to pause in the middle of a computation to allow the user time to view the display. `sit-for` pauses and updates the screen, and returns immediately if input comes in, while `sleep-for` pauses without updating the screen.

`sit-for` *seconds* **&optional** *nodisp* [Function]

This function performs redisplay (provided there is no pending input from the user), then waits *seconds* seconds, or until input is available. The usual purpose of `sit-for` is to give the user time to read text that you display. The value is `t` if `sit-for` waited the full time with no input arriving (see Section 2.8.6 [Event Input Misc], page 45). Otherwise, the value is `nil`.

The argument *seconds* need not be an integer. If it is a floating point number, `sit-for` waits for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down.

The expression `(sit-for 0)` is equivalent to `(redisplay)`, i.e., it requests a redisplay, without any delay, if there is no pending input. See Section 11.2 [Forcing Redisplay], page 111.

If *nodisp* is non-`nil`, then `sit-for` does not redisplay, but it still returns as soon as input is available (or when the timeout elapses).

In batch mode (see ⟨undefined⟩ [Batch Mode], page ⟨undefined⟩), `sit-for` cannot be interrupted, even by input from the standard input descriptor. It is thus equivalent to `sleep-for`, which is described below.

It is also possible to call `sit-for` with three arguments, as `(sit-for` *`seconds millisec nodisp`*`)`, but that is considered obsolete.

`sleep-for` *seconds* **&optional** *millisec* [Function]

This function simply pauses for *seconds* seconds without updating the display. It pays no attention to available input. It returns `nil`.

The argument *seconds* need not be an integer. If it is a floating point number, `sleep-for` waits for a fractional number of seconds. Some systems support only a whole number of seconds; on these systems, *seconds* is rounded down.

The optional argument *millisec* specifies an additional waiting period measured in milliseconds. This adds to the period specified by *seconds*. If the system doesn't support waiting fractions of a second, you get an error if you specify nonzero *millisec*.

Use `sleep-for` when you wish to guarantee a delay.

See ⟨undefined⟩ [Time of Day], page ⟨undefined⟩, for functions to get the current time.

## 2.11 Quitting

Typing `C-g` while a Lisp function is running causes Emacs to *quit* whatever it is doing. This means that control returns to the innermost active command loop.

Typing `C-g` while the command loop is waiting for keyboard input does not cause a quit; it acts as an ordinary input character. In the simplest case, you cannot tell the difference, because `C-g` normally runs the command `keyboard-quit`, whose effect is to quit. However, when `C-g` follows a prefix key, they combine to form an undefined key. The effect is to cancel the prefix key as well as any prefix argument.

In the minibuffer, `C-g` has a different definition: it aborts out of the minibuffer. This means, in effect, that it exits the minibuffer and then quits. (Simply quitting would return to the command loop *within* the minibuffer.) The reason why `C-g` does not quit directly when the command reader is reading input is so that its meaning can be redefined in the minibuffer in this way. `C-g` following a prefix key is not redefined in the minibuffer, and it has its normal effect of canceling the prefix key and prefix argument. This too would not be possible if `C-g` always quit directly.

When `C-g` does directly quit, it does so by setting the variable `quit-flag` to `t`. Emacs checks this variable at appropriate times and quits if it is not `nil`. Setting `quit-flag` non-`nil` in any way thus causes a quit.

At the level of C code, quitting cannot happen just anywhere; only at the special places that check `quit-flag`. The reason for this is that quitting at other places might leave an inconsistency in Emacs's internal state. Because quitting is delayed until a safe place, quitting cannot make Emacs crash.

Certain functions such as `read-key-sequence` or `read-quoted-char` prevent quitting entirely even though they wait for input. Instead of quitting, `C-g` serves as the requested input. In the case of `read-key-sequence`, this serves to bring about the special behavior of `C-g` in the command loop. In the case of `read-quoted-char`, this is so that `C-q` can be used to quote a `C-g`.

You can prevent quitting for a portion of a Lisp function by binding the variable `inhibit-quit` to a non-`nil` value. Then, although `C-g` still sets `quit-flag` to `t` as usual, the usual result of this—a quit—is prevented. Eventually, `inhibit-quit` will become `nil` again, such as when its binding is unwound at the end of a `let` form. At that time, if `quit-flag` is still non-`nil`, the requested quit happens immediately. This behavior is ideal when you wish to make sure that quitting does not happen within a "critical section" of the program.

In some functions (such as `read-quoted-char`), `C-g` is handled in a special way that does not involve quitting. This is done by reading the input with `inhibit-quit` bound to `t`, and setting `quit-flag` to `nil` before `inhibit-quit` becomes `nil` again. This excerpt from the definition of `read-quoted-char` shows how this is done; it also shows that normal quitting is permitted after the first character of input.

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((message-log-max nil) done (first t) (code 0) char)
    (while (not done)
      (let ((inhibit-quit first)
            ...)
```

```
      (and prompt (message "%s-" prompt))
      (setq char (read-event))
      (if inhibit-quit (setq quit-flag nil)))
    ...set the variable code...)
  code))
```

`quit-flag` [Variable]

If this variable is non-`nil`, then Emacs quits immediately, unless `inhibit-quit` is non-`nil`. Typing `C-g` ordinarily sets `quit-flag` non-`nil`, regardless of `inhibit-quit`.

`inhibit-quit` [Variable]

This variable determines whether Emacs should quit when `quit-flag` is set to a value other than `nil`. If `inhibit-quit` is non-`nil`, then `quit-flag` has no special effect.

`with-local-quit` *body...* [Macro]

This macro executes *body* forms in sequence, but allows quitting, at least locally, within *body* even if `inhibit-quit` was non-`nil` outside this construct. It returns the value of the last form in *body*, unless exited by quitting, in which case it returns `nil`.

If `inhibit-quit` is `nil` on entry to `with-local-quit`, it only executes the *body*, and setting `quit-flag` causes a normal quit. However, if `inhibit-quit` is non-`nil` so that ordinary quitting is delayed, a non-`nil` `quit-flag` triggers a special kind of local quit. This ends the execution of *body* and exits the `with-local-quit` body with `quit-flag` still non-`nil`, so that another (ordinary) quit will happen as soon as that is allowed. If `quit-flag` is already non-`nil` at the beginning of *body*, the local quit happens immediately and the body doesn't execute at all.

This macro is mainly useful in functions that can be called from timers, process filters, process sentinels, `pre-command-hook`, `post-command-hook`, and other places where `inhibit-quit` is normally bound to `t`.

`keyboard-quit` [Command]

This function signals the `quit` condition with `(signal 'quit nil)`. This is the same thing that quitting does. (See `signal` in ⟨undefined⟩ [Errors], page ⟨undefined⟩.)

You can specify a character other than `C-g` to use for quitting. See the function `set-input-mode` in ⟨undefined⟩ [Terminal Input], page ⟨undefined⟩.

## 2.12 Prefix Command Arguments

Most Emacs commands can use a *prefix argument*, a number specified before the command itself. (Don't confuse prefix arguments with prefix keys.) The prefix argument is at all times represented by a value, which may be `nil`, meaning there is currently no prefix argument. Each command may use the prefix argument or ignore it.

There are two representations of the prefix argument: *raw* and *numeric*. The editor command loop uses the raw representation internally, and so do the Lisp variables that store the information, but commands can request either representation.

Here are the possible values of a raw prefix argument:

- `nil`, meaning there is no prefix argument. Its numeric value is 1, but numerous commands make a distinction between `nil` and the integer 1.

- An integer, which stands for itself.
- A list of one element, which is an integer. This form of prefix argument results from one or a succession of `C-u`s with no digits. The numeric value is the integer in the list, but some commands make a distinction between such a list and an integer alone.
- The symbol `-`. This indicates that `M--` or `C-u -` was typed, without following digits. The equivalent numeric value is −1, but some commands make a distinction between the integer −1 and the symbol `-`.

We illustrate these possibilities by calling the following function with various prefixes:

```
(defun display-prefix (arg)
  "Display the value of the raw prefix arg."
  (interactive "P")
  (message "%s" arg))
```

Here are the results of calling `display-prefix` with various raw prefix arguments:

```
          M-x display-prefix  ⊣ nil

C-u       M-x display-prefix  ⊣ (4)

C-u C-u   M-x display-prefix  ⊣ (16)

C-u 3     M-x display-prefix  ⊣ 3

M-3       M-x display-prefix  ⊣ 3       ; (Same as C-u 3.)

C-u -     M-x display-prefix  ⊣ -

M--       M-x display-prefix  ⊣ -       ; (Same as C-u -.)

C-u - 7   M-x display-prefix  ⊣ -7

M-- 7     M-x display-prefix  ⊣ -7      ; (Same as C-u -7.)
```

Emacs uses two variables to store the prefix argument: `prefix-arg` and `current-prefix-arg`. Commands such as `universal-argument` that set up prefix arguments for other commands store them in `prefix-arg`. In contrast, `current-prefix-arg` conveys the prefix argument to the current command, so setting it has no effect on the prefix arguments for future commands.

Normally, commands specify which representation to use for the prefix argument, either numeric or raw, in the `interactive` specification. (See Section 2.2.1 [Using Interactive], page 12.) Alternatively, functions may look at the value of the prefix argument directly in the variable `current-prefix-arg`, but this is less clean.

`prefix-numeric-value` *arg* [Function]

This function returns the numeric meaning of a valid raw prefix argument value, *arg*. The argument may be a symbol, a number, or a list. If it is `nil`, the value 1 is returned; if it is `-`, the value −1 is returned; if it is a number, that number is returned; if it is a list, the CAR of that list (which should be a number) is returned.

`current-prefix-arg` [Variable]
: This variable holds the raw prefix argument for the *current* command. Commands may examine it directly, but the usual method for accessing it is with `(interactive "P")`.

`prefix-arg` [Variable]
: The value of this variable is the raw prefix argument for the *next* editing command. Commands such as `universal-argument` that specify prefix arguments for the following command work by setting this variable.

`last-prefix-arg` [Variable]
: The raw prefix argument value used by the previous command.

The following commands exist to set up prefix arguments for the following command. Do not call them for any other reason.

`universal-argument` [Command]
: This command reads input and specifies a prefix argument for the following command. Don't call this command yourself unless you know what you are doing.

`digit-argument` *arg* [Command]
: This command adds to the prefix argument for the following command. The argument *arg* is the raw prefix argument as it was before this command; it is used to compute the updated prefix argument. Don't call this command yourself unless you know what you are doing.

`negative-argument` *arg* [Command]
: This command adds to the numeric argument for the next command. The argument *arg* is the raw prefix argument as it was before this command; its value is negated to form the new prefix argument. Don't call this command yourself unless you know what you are doing.

## 2.13 Recursive Editing

The Emacs command loop is entered automatically when Emacs starts up. This top-level invocation of the command loop never exits; it keeps running as long as Emacs does. Lisp programs can also invoke the command loop. Since this makes more than one activation of the command loop, we call it *recursive editing*. A recursive editing level has the effect of suspending whatever command invoked it and permitting the user to do arbitrary editing before resuming that command.

The commands available during recursive editing are the same ones available in the top-level editing loop and defined in the keymaps. Only a few special commands exit the recursive editing level; the others return to the recursive editing level when they finish. (The special commands for exiting are always available, but they do nothing when recursive editing is not in progress.)

All command loops, including recursive ones, set up all-purpose error handlers so that an error in a command run from the command loop will not exit the loop.

Minibuffer input is a special kind of recursive editing. It has a few special wrinkles, such as enabling display of the minibuffer and the minibuffer window, but fewer than you might

suppose. Certain keys behave differently in the minibuffer, but that is only because of the minibuffer's local map; if you switch windows, you get the usual Emacs commands.

To invoke a recursive editing level, call the function `recursive-edit`. This function contains the command loop; it also contains a call to `catch` with tag `exit`, which makes it possible to exit the recursive editing level by throwing to `exit` (see ⟨undefined⟩ [Catch and Throw], page ⟨undefined⟩). If you throw a value other than `t`, then `recursive-edit` returns normally to the function that called it. The command `C-M-c` (`exit-recursive-edit`) does this. Throwing a `t` value causes `recursive-edit` to quit, so that control returns to the command loop one level up. This is called *aborting*, and is done by `C-]` (`abort-recursive-edit`).

Most applications should not use recursive editing, except as part of using the minibuffer. Usually it is more convenient for the user if you change the major mode of the current buffer temporarily to a special major mode, which should have a command to go back to the previous mode. (The `e` command in Rmail uses this technique.) Or, if you wish to give the user different text to edit "recursively", create and select a new buffer in a special mode. In this mode, define a command to complete the processing and go back to the previous buffer. (The `m` command in Rmail does this.)

Recursive edits are useful in debugging. You can insert a call to `debug` into a function definition as a sort of breakpoint, so that you can look around when the function gets there. `debug` invokes a recursive edit but also provides the other features of the debugger.

Recursive editing levels are also used when you type `C-r` in `query-replace` or use `C-x q` (`kbd-macro-query`).

`recursive-edit` [Command]

This function invokes the editor command loop. It is called automatically by the initialization of Emacs, to let the user begin editing. When called from a Lisp program, it enters a recursive editing level.

If the current buffer is not the same as the selected window's buffer, `recursive-edit` saves and restores the current buffer. Otherwise, if you switch buffers, the buffer you switched to is current after `recursive-edit` returns.

In the following example, the function `simple-rec` first advances point one word, then enters a recursive edit, printing out a message in the echo area. The user can then do any editing desired, and then type `C-M-c` to exit and continue executing `simple-rec`.

```
(defun simple-rec ()
  (forward-word 1)
  (message "Recursive edit in progress")
  (recursive-edit)
  (forward-word 1))
     ⇒ simple-rec
(simple-rec)
     ⇒ nil
```

`exit-recursive-edit` [Command]

This function exits from the innermost recursive edit (including minibuffer input). Its definition is effectively `(throw 'exit nil)`.

`abort-recursive-edit` [Command]
: This function aborts the command that requested the innermost recursive edit (including minibuffer input), by signaling `quit` after exiting the recursive edit. Its definition is effectively `(throw 'exit t)`. See Section 34.1 [Quitting], page 717.

`top-level` [Command]
: This function exits all recursive editing levels; it does not return a value, as it jumps completely out of any computation directly back to the main command loop.

`recursion-depth` [Function]
: This function returns the current depth of recursive edits. When no recursive edit is active, it returns 0.

## 2.14 Disabling Commands

*Disabling a command* marks the command as requiring user confirmation before it can be executed. Disabling is used for commands which might be confusing to beginning users, to prevent them from using the commands by accident.

The low-level mechanism for disabling a command is to put a non-`nil` `disabled` property on the Lisp symbol for the command. These properties are normally set up by the user's init file (see Section 33.4 [Init File], page 711) with Lisp expressions such as this:

```
(put 'upcase-region 'disabled t)
```

For a few commands, these properties are present by default (you can remove them in your init file if you wish).

If the value of the `disabled` property is a string, the message saying the command is disabled includes that string. For example:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

See Section "Disabling" in *The GNU Emacs Manual*, for the details on what happens when a disabled command is invoked interactively. Disabling a command has no effect on calling it as a function from Lisp programs.

`enable-command` *command* [Command]
: Allow *command* (a symbol) to be executed without special confirmation from now on, and alter the user's init file (see Section 33.4 [Init File], page 711) so that this will apply to future sessions.

`disable-command` *command* [Command]
: Require special confirmation to execute *command* from now on, and alter the user's init file so that this will apply to future sessions.

`disabled-command-function` [Variable]
: The value of this variable should be a function. When the user invokes a disabled command interactively, this function is called instead of the disabled command. It can use `this-command-keys` to determine what the user typed to run the command, and thus find the command itself.

  The value may also be `nil`. Then all commands work normally, even disabled ones.

  By default, the value is a function that asks the user whether to proceed.

## 2.15 Command History

The command loop keeps a history of the complex commands that have been executed, to make it convenient to repeat these commands. A *complex command* is one for which the interactive argument reading uses the minibuffer. This includes any `M-x` command, any `M-:` command, and any command whose `interactive` specification reads an argument from the minibuffer. Explicit use of the minibuffer during the execution of the command itself does not cause the command to be considered complex.

`command-history` [Variable]

This variable's value is a list of recent complex commands, each represented as a form to evaluate. It continues to accumulate all complex commands for the duration of the editing session, but when it reaches the maximum size (see Section 5.5 [Minibuffer History], page 74), the oldest elements are deleted as new ones are added.

```
command-history
⇒ ((switch-to-buffer "chistory.texi")
    (describe-key "^X^[")
    (visit-tags-table "~/emacs/src/")
    (find-tag "repeat-complex-command"))
```

This history list is actually a special case of minibuffer history (see Section 5.5 [Minibuffer History], page 74), with one special twist: the elements are expressions rather than strings.

There are a number of commands devoted to the editing and recall of previous commands. The commands `repeat-complex-command`, and `list-command-history` are described in the user manual (see Section "Repetition" in *The GNU Emacs Manual*). Within the minibuffer, the usual minibuffer history commands are available.

## 2.16 Keyboard Macros

A *keyboard macro* is a canned sequence of input events that can be considered a command and made the definition of a key. The Lisp representation of a keyboard macro is a string or vector containing the events. Don't confuse keyboard macros with Lisp macros (see ⟨undefined⟩ [Macros], page ⟨undefined⟩).

`execute-kbd-macro` *kbdmacro* **&optional** *count loopfunc* [Function]

This function executes *kbdmacro* as a sequence of events. If *kbdmacro* is a string or vector, then the events in it are executed exactly as if they had been input by the user. The sequence is *not* expected to be a single key sequence; normally a keyboard macro definition consists of several key sequences concatenated.

If *kbdmacro* is a symbol, then its function definition is used in place of *kbdmacro*. If that is another symbol, this process repeats. Eventually the result should be a string or vector. If the result is not a symbol, string, or vector, an error is signaled.

The argument *count* is a repeat count; *kbdmacro* is executed that many times. If *count* is omitted or `nil`, *kbdmacro* is executed once. If it is 0, *kbdmacro* is executed over and over until it encounters an error or a failing search.

If *loopfunc* is non-`nil`, it is a function that is called, without arguments, prior to each iteration of the macro. If *loopfunc* returns `nil`, then this stops execution of the macro.

See Section 2.8.2 [Reading One Event], page 41, for an example of using `execute-kbd-macro`.

`executing-kbd-macro` [Variable]
This variable contains the string or vector that defines the keyboard macro that is currently executing. It is `nil` if no macro is currently executing. A command can test this variable so as to behave differently when run from an executing macro. Do not set this variable yourself.

`defining-kbd-macro` [Variable]
This variable is non-`nil` if and only if a keyboard macro is being defined. A command can test this variable so as to behave differently while a macro is being defined. The value is `append` while appending to the definition of an existing macro. The commands `start-kbd-macro`, `kmacro-start-macro` and `end-kbd-macro` set this variable—do not set it yourself.

The variable is always local to the current terminal and cannot be buffer-local. See Section 18.2 [Multiple Terminals], page 342.

`last-kbd-macro` [Variable]
This variable is the definition of the most recently defined keyboard macro. Its value is a string or vector, or `nil`.

The variable is always local to the current terminal and cannot be buffer-local. See Section 18.2 [Multiple Terminals], page 342.

`kbd-macro-termination-hook` [Variable]
This normal hook is run when a keyboard macro terminates, regardless of what caused it to terminate (reaching the macro end or an error which ended the macro prematurely).

# 3 Entering and Exiting Emacs

This chapter explains how to enter Emacs, and how to exit it.

## 3.1 Entering Emacs

The usual way to invoke Emacs is with the shell command `emacs`. From a terminal window running in the X Window System, you can run Emacs in the background with `emacs &`; this way, Emacs won't tie up the terminal window, so you can use it to run other shell commands.

When Emacs starts up, the initial frame displays a special buffer named '`*GNU Emacs*`'. This *startup screen* contains information about Emacs and *links* to common tasks that are useful for beginning users. For instance, activating the '`Emacs Tutorial`' link opens the Emacs tutorial; this does the same thing as the command `C-h t` (`help-with-tutorial`). To activate a link, either move point onto it and type RET, or click on it with `mouse-1` (the left mouse button).

Using a command line argument, you can tell Emacs to visit one or more files as soon as it starts up. For example, `emacs foo.txt` starts Emacs with a buffer displaying the contents of the file '`foo.txt`'. This feature exists mainly for compatibility with other editors, which are designed to be launched from the shell for short editing sessions. If you call Emacs this way, the initial frame is split into two windows—one showing the specified file, and the other showing the startup screen. See Chapter 17 [Windows], page 289.

Generally, it is unnecessary and wasteful to start Emacs afresh each time you want to edit a file. The recommended way to use Emacs is to start it just once, just after you log in, and do all your editing in the same Emacs session. See Chapter 15 [Files], page 230, for information on visiting more than one file. If you use Emacs this way, the Emacs session accumulates valuable context, such as the kill ring, registers, undo history, and mark ring data, which together make editing more convenient. These features are described later in the manual.

To edit a file from another program while Emacs is running, you can use the `emacsclient` helper program to open a file in the existing Emacs session. See Section 31.4 [Emacs Server], page 665.

Emacs accepts other command line arguments that tell it to load certain Lisp files, where to put the initial frame, and so forth. See Appendix E [Emacs Invocation], page 754.

If the variable `inhibit-startup-screen` is non-`nil`, Emacs does not display the startup screen. In that case, if one or more files were specified on the command line, Emacs simply displays those files; otherwise, it displays a buffer named '`*scratch*`', which can be used to evaluate Emacs Lisp expressions interactively. See Section 24.10 [Lisp Interaction], page 552. You can set the variable `inhibit-startup-screen` using the Customize facility (see Section 33.1 [Easy Customization], page 686), or by editing your initialization file (see Section 33.4 [Init File], page 711).[1]

You can also force Emacs to display a file or directory at startup by setting the variable `initial-buffer-choice` to a non-`nil` value. (In that case, even if you specify one or

[1] Setting `inhibit-startup-screen` in '`site-start.el`' doesn't work, because the startup screen is set up before reading '`site-start.el`'. See Section 33.4 [Init File], page 711, for information about '`site-start.el`'.

more files on the command line, Emacs opens but does not display them.) The value of `initial-buffer-choice` should be the name of the desired file or directory.

## 3.2 Exiting Emacs

`C-x C-c` Kill Emacs (`save-buffers-kill-terminal`).

`C-z` On a text terminal, suspend Emacs; on a graphical display, "minimize" the selected frame (`suspend-emacs`).

*Killing* Emacs means terminating the Emacs program. To do this, type `C-x C-c` (`save-buffers-kill-terminal`). A two-character key sequence is used to make it harder to type by accident. If there are any modified file-visiting buffers when you type `C-x C-c`, Emacs first offers to save these buffers. If you do not save them all, it asks for confirmation again, since the unsaved changes will be lost. Emacs also asks for confirmation if any subprocesses are still running, since killing Emacs will also kill the subprocesses (see Section 31.3 [Shell], page 655).

`C-x C-c` behaves specially if you are using Emacs as a server. If you type it from a "client frame", it closes the client connection. See Section 31.4 [Emacs Server], page 665.

Emacs can, optionally, record certain session information when you kill it, such as the files you were visiting at the time. This information is then available the next time you start Emacs. See Section 31.8 [Saving Emacs Sessions], page 675.

If the value of the variable `confirm-kill-emacs` is non-`nil`, `C-x C-c` assumes that its value is a predicate function, and calls that function. If the result of the function call is non-`nil`, the session is killed, otherwise Emacs continues to run. One convenient function to use as the value of `confirm-kill-emacs` is the function `yes-or-no-p`. The default value of `confirm-kill-emacs` is `nil`.

To kill Emacs without being prompted about saving, type `M-x kill-emacs`.

`C-z` runs the command `suspend-frame`. On a graphical display, this command *minimizes* (or *iconifies*) the selected Emacs frame, hiding it in a way that lets you bring it back later (exactly how this hiding occurs depends on the window system). On a text terminal, the `C-z` command *suspends* Emacs, stopping the program temporarily and returning control to the parent process (usually a shell); in most shells, you can resume Emacs after suspending it with the shell command `%emacs`.

Text terminals usually listen for certain special characters whose meaning is to kill or suspend the program you are running. **This terminal feature is turned off while you are in Emacs.** The meanings of `C-z` and `C-x C-c` as keys in Emacs were inspired by the use of `C-z` and `C-c` on several operating systems as the characters for stopping or killing a program, but that is their only relationship with the operating system. You can customize these keys to run any commands of your choice (see Section 33.3.1 [Keymaps], page 703).

# 4 Basic Editing Commands

Here we explain the basics of how to enter text, make corrections, and save the text in a file. If this material is new to you, we suggest you first run the Emacs learn-by-doing tutorial, by typing `C-h t` (`help-with-tutorial`).

## 4.1 Inserting Text

You can insert an ordinary *graphic character* (e.g., '`a`', '`B`', '`3`', and '`=`') by typing the associated key. This adds the character to the buffer at point. Insertion moves point forward, so that point remains just after the inserted text. See Section 1.1 [Point], page 6.

To end a line and start a new one, type RET (`newline`). (The RET key may be labeled RETURN or ENTER on your keyboard, but we refer to it as RET in this manual.) This command inserts a newline character into the buffer. If point is at the end of the line, the effect is to create a new blank line after it; if point is in the middle of a line, the line is split at that position.

As we explain later in this manual, you can change the way Emacs handles text insertion by turning on *minor modes*. For instance, the minor mode called Auto Fill mode splits lines automatically when they get too long (see Section 22.11 [Filling], page 473). The minor mode called Overwrite mode causes inserted characters to replace (overwrite) existing text, instead of shoving it to the right. See Section 20.3 [Minor Modes], page 413.

Only graphic characters can be inserted by typing the associated key; other keys act as editing commands and do not insert themselves. For instance, `DEL` runs the command `delete-backward-char` by default (some modes bind it to a different command); it does not insert a literal '`DEL`' character (ASCII character code 127).

To insert a non-graphic character, or a character that your keyboard does not support, first *quote* it by typing `C-q` (`quoted-insert`). There are two ways to use `C-q`:

- `C-q` followed by any non-graphic character (even `C-g`) inserts that character. For instance, `C-q DEL` inserts a literal '`DEL`' character.
- `C-q` followed by a sequence of octal digits inserts the character with the specified octal character code. You can use any number of octal digits; any non-digit terminates the sequence. If the terminating character is RET, that RET serves only to terminate the sequence. Any other non-digit terminates the sequence and then acts as normal input—thus, `C-q 1 0 1 B` inserts '`AB`'.

  The use of octal sequences is disabled in ordinary non-binary Overwrite mode, to give you a convenient way to insert a digit instead of overwriting with it.

To use decimal or hexadecimal instead of octal, set the variable `read-quoted-char-radix` to 10 or 16. If the radix is 16, the letters `a` to `f` serve as part of a character code, just like digits. Case is ignored.

Alternatively, you can use the command `C-x 8 RET` (`insert-char`). This prompts for the Unicode name or code-point of a character, using the minibuffer. If you enter a name, the command provides completion (see Section 5.4 [Completion], page 70). If you enter a code-point, it should be as a hexadecimal number (the convention for Unicode), or a number with a specified radix, e.g., `#o23072` (octal); See Section "Integer Basics" in *The Emacs Lisp Reference Manual*. The command then inserts the corresponding character into

the buffer. For example, both of the following insert the infinity sign (Unicode code-point U+221E):

```
C-x 8 RET infinity RET
C-x 8 RET 221e RET
```

A numeric argument to C-q or C-x 8 RET specifies how many copies of the character to insert (see Section 4.10 [Arguments], page 65).

## 4.2 Changing the Location of Point

To do more than insert characters, you have to know how to move point (see Section 1.1 [Point], page 6). The keyboard commands C-f, C-b, C-n, and C-p move point to the right, left, down, and up, respectively. You can also move point using the *arrow keys* present on most keyboards: RIGHT, LEFT, DOWN, and UP; however, many Emacs users find that it is slower to use the arrow keys than the control keys, because you need to move your hand to the area of the keyboard where those keys are located.

You can also click the left mouse button to move point to the position clicked. Emacs also provides a variety of additional keyboard commands that move point in more sophisticated ways.

C-f
: Move forward one character (forward-char).

RIGHT
: This command (right-char) behaves like C-f, with one exception: when editing right-to-left scripts such as Arabic, it instead moves *backward* if the current paragraph is a right-to-left paragraph. See Section 19.20 [Bidirectional Editing], page 394.

C-b
: Move backward one character (backward-char).

LEFT
: This command (left-char) behaves like C-b, except it moves *forward* if the current paragraph is right-to-left. See Section 19.20 [Bidirectional Editing], page 394.

C-n
DOWN
: Move down one screen line (next-line). This command attempts to keep the horizontal position unchanged, so if you start in the middle of one line, you move to the middle of the next.

C-p
UP
: Move up one screen line (previous-line). This command preserves position within the line, like C-n.

C-a
HOME
: Move to the beginning of the line (move-beginning-of-line).

C-e
END
: Move to the end of the line (move-end-of-line).

M-f
: Move forward one word (forward-word).

C-RIGHT
M-RIGHT
: This command (right-word) behaves like M-f, except it moves *backward* by one word if the current paragraph is right-to-left. See Section 19.20 [Bidirectional Editing], page 394.

`M-b` Move backward one word (`backward-word`).

`C-LEFT`
`M-LEFT` This command (`left-word`) behaves like `M-f`, except it moves *forward* by one word if the current paragraph is right-to-left. See Section 19.20 [Bidirectional Editing], page 394.

`M-r` Without moving the text on the screen, reposition point on the left margin of the center-most text line of the window; on subsequent consecutive invocations, move point to the left margin of the top-most line, the bottom-most line, and so forth, in cyclic order (`move-to-window-line-top-bottom`).

A numeric argument says which screen line to place point on, counting downward from the top of the window (zero means the top line). A negative argument counts lines up from the bottom (−1 means the bottom line). See Section 4.10 [Arguments], page 65, for more information on numeric arguments.

`M-<` Move to the top of the buffer (`beginning-of-buffer`). With numeric argument $n$, move to $n/10$ of the way from the top.

`M->` Move to the end of the buffer (`end-of-buffer`).

`C-v`
PAGEDOWN
NEXT Scroll the display one screen forward, and move point onscreen if necessary (`scroll-up-command`). See ⟨undefined⟩ [Scrolling], page ⟨undefined⟩.

`M-v`
PAGEUP
PRIOR Scroll one screen backward, and move point onscreen if necessary (`scroll-down-command`). See ⟨undefined⟩ [Scrolling], page ⟨undefined⟩.

`M-g c` Read a number $n$ and move point to buffer position $n$. Position 1 is the beginning of the buffer.

`M-g M-g`
`M-g g` Read a number $n$ and move point to the beginning of line number $n$ (`goto-line`). Line 1 is the beginning of the buffer. If point is on or just after a number in the buffer, that is the default for $n$. Just type RET in the minibuffer to use it. You can also specify $n$ by giving `M-g M-g` a numeric prefix argument. See ⟨undefined⟩ [Select Buffer], page ⟨undefined⟩, for the behavior of `M-g M-g` when you give it a plain prefix argument.

`M-g TAB` Read a number $n$ and move to column $n$ in the current line. Column 0 is the leftmost column. If called with a prefix argument, move to the column number specified by the argument's numeric value.

`C-x C-n` Use the current column of point as the *semipermanent goal column* for `C-n` and `C-p` (`set-goal-column`). When a semipermanent goal column is in effect, those commands always try to move to this column, or as close as possible to it, after moving vertically. The goal column remains in effect until canceled.

`C-u C-x C-n`
Cancel the goal column. Henceforth, `C-n` and `C-p` try to preserve the horizontal position, as usual.

When a line of text in the buffer is longer than the width of the window, Emacs usually displays it on two or more *screen lines*. For convenience, `C-n` and `C-p` move point by screen lines, as do the equivalent keys DOWN and UP. You can force these commands to move according to *logical lines* (i.e., according to the text lines in the buffer) by setting the variable `line-move-visual` to `nil`; if a logical line occupies multiple screen lines, the cursor then skips over the additional screen lines. For details, see Section 4.8 [Continuation Lines], page 63. See Section 33.2 [Variables], page 694, for how to set variables such as `line-move-visual`.

Unlike `C-n` and `C-p`, most of the Emacs commands that work on lines work on *logical* lines. For instance, `C-a` (`move-beginning-of-line`) and `C-e` (`move-end-of-line`) respectively move to the beginning and end of the logical line. Whenever we encounter commands that work on screen lines, such as `C-n` and `C-p`, we will point these out.

When `line-move-visual` is `nil`, you can also set the variable `track-eol` to a non-`nil` value. Then `C-n` and `C-p`, when starting at the end of the logical line, move to the end of the next logical line. Normally, `track-eol` is `nil`.

`C-n` normally stops at the end of the buffer when you use it on the last line in the buffer. However, if you set the variable `next-line-add-newlines` to a non-`nil` value, `C-n` on the last line of a buffer creates an additional line at the end and moves down into it.

## 4.3 Erasing Text

DEL
BACKSPACE
: Delete the character before point, or the region if it is active (`delete-backward-char`).

DELETE
: Delete the character after point, or the region if it is active (`delete-forward-char`).

`C-d`
: Delete the character after point (`delete-char`).

`C-k`
: Kill to the end of the line (`kill-line`).

`M-d`
: Kill forward to the end of the next word (`kill-word`).

`M-DEL`
: Kill back to the beginning of the previous word (`backward-kill-word`).

The DEL (`delete-backward-char`) command removes the character before point, moving the cursor and the characters after it backwards. If point was at the beginning of a line, this deletes the preceding newline, joining this line to the previous one.

If, however, the region is active, DEL instead deletes the text in the region. See Chapter 8 [Mark], page 89, for a description of the region.

On most keyboards, DEL is labeled BACKSPACE, but we refer to it as DEL in this manual. (Do not confuse DEL with the DELETE key; we will discuss DELETE momentarily.) On some text terminals, Emacs may not recognize the DEL key properly. See Section 34.2.1 [DEL Does Not Delete], page 718, if you encounter this problem.

The DELETE (`delete-forward-char`) command deletes in the "opposite direction": it deletes the character after point, i.e., the character under the cursor. If point was at the

end of a line, this joins the following line onto this one. Like DEL, it deletes the text in the region if the region is active (see Chapter 8 [Mark], page 89).

`C-d` (`delete-char`) deletes the character after point, similar to DELETE, but regardless of whether the region is active.

See Section 22.6 [Deletion], page 460, for more detailed information about the above deletion commands.

`C-k` (`kill-line`) erases (kills) a line at a time. If you type `C-k` at the beginning or middle of a line, it kills all the text up to the end of the line. If you type `C-k` at the end of a line, it joins that line with the following line.

See Chapter 9 [Killing], page 95, for more information about `C-k` and related commands.

## 4.4 Undoing Changes

`C-/` Undo one entry of the undo records—usually, one command worth (`undo`).

`C-x u`
`C-_` The same.

Emacs records a list of changes made in the buffer text, so you can undo recent changes. This is done using the `undo` command, which is bound to `C-/` (as well as `C-x u` and `C-_`). Normally, this command undoes the last change, moving point back to where it was before the change. The undo command applies only to changes in the buffer; you can't use it to undo cursor motion.

Although each editing command usually makes a separate entry in the undo records, very simple commands may be grouped together. Sometimes, an entry may cover just part of a complex command.

If you repeat `C-/` (or its aliases), each repetition undoes another, earlier change, back to the limit of the undo information available. If all recorded changes have already been undone, the undo command displays an error message and does nothing.

To learn more about the `undo` command, see Section 22.9 [Undo], page 469.

## 4.5 Files

Text that you insert in an Emacs buffer lasts only as long as the Emacs session. To keep any text permanently, you must put it in a *file*.

Suppose there is a file named '`test.emacs`' in your home directory. To begin editing this file in Emacs, type

```
C-x C-f test.emacs RET
```

Here the file name is given as an *argument* to the command `C-x C-f` (`find-file`). That command uses the *minibuffer* to read the argument, and you type RET to terminate the argument (see Chapter 5 [Minibuffer], page 68).

Emacs obeys this command by *visiting* the file: it creates a buffer, copies the contents of the file into the buffer, and then displays the buffer for editing. If you alter the text, you can *save* the new text in the file by typing `C-x C-s` (`save-buffer`). This copies the altered buffer contents back into the file '`test.emacs`', making them permanent. Until you save, the changed text exists only inside Emacs, and the file '`test.emacs`' is unaltered.

To create a file, just visit it with `C-x C-f` as if it already existed. This creates an empty buffer, in which you can insert the text you want to put in the file. Emacs actually creates the file the first time you save this buffer with `C-x C-s`.

To learn more about using files in Emacs, see Chapter 15 [Files], page 230.

## 4.6 Help

If you forget what a key does, you can find out by typing `C-h k` (`describe-key`), followed by the key of interest; for example, `C-h k C-n` tells you what `C-n` does.

The prefix key `C-h` stands for "help". The key F1 serves as an alias for `C-h`. Apart from `C-h k`, there are many other help commands providing different kinds of help.

See ⟨undefined⟩ [Help], page ⟨undefined⟩, for details.

## 4.7 Blank Lines

Here are special commands and techniques for inserting and deleting blank lines.

`C-o` Insert a blank line after the cursor (`open-line`).

`C-x C-o` Delete all but one of many consecutive blank lines (`delete-blank-lines`).

We have seen how RET (`newline`) starts a new line of text. However, it may be easier to see what you are doing if you first make a blank line and then insert the desired text into it. This is easy to do using the key `C-o` (`open-line`), which inserts a newline after point but leaves point in front of the newline. After `C-o`, type the text for the new line.

You can make several blank lines by typing `C-o` several times, or by giving it a numeric argument specifying how many blank lines to make. See Section 4.10 [Arguments], page 65, for how. If you have a fill prefix, the `C-o` command inserts the fill prefix on the new line, if typed at the beginning of a line. See ⟨undefined⟩ [Fill Prefix], page ⟨undefined⟩.

The easy way to get rid of extra blank lines is with the command `C-x C-o` (`delete-blank-lines`). If point lies within a run of several blank lines, `C-x C-o` deletes all but one of them. If point is on a single blank line, `C-x C-o` deletes it. If point is on a nonblank line, `C-x C-o` deletes all following blank lines, if any exists.

## 4.8 Continuation Lines

Sometimes, a line of text in the buffer—a *logical line*—is too long to fit in the window, and Emacs displays it as two or more *screen lines*. This is called *line wrapping* or *continuation*, and the long logical line is called a *continued line*. On a graphical display, Emacs indicates line wrapping with small bent arrows in the left and right window fringes. On a text terminal, Emacs indicates line wrapping by displaying a '\' character at the right margin.

Most commands that act on lines act on logical lines, not screen lines. For instance, `C-k` kills a logical line. As described earlier, `C-n` (`next-line`) and `C-p` (`previous-line`) are special exceptions: they move point down and up, respectively, by one screen line (see Section 4.2 [Moving Point], page 59).

Emacs can optionally *truncate* long logical lines instead of continuing them. This means that every logical line occupies a single screen line; if it is longer than the width of the window, the rest of the line is not displayed. On a graphical display, a truncated line is

indicated by a small straight arrow in the right fringe; on a text terminal, it is indicated by a '$' character in the right margin. See ⟨undefined⟩ [Line Truncation], page ⟨undefined⟩.

By default, continued lines are wrapped at the right window edge. Since the wrapping may occur in the middle of a word, continued lines can be difficult to read. The usual solution is to break your lines before they get too long, by inserting newlines. If you prefer, you can make Emacs insert a newline automatically when a line gets too long, by using Auto Fill mode. See Section 22.11 [Filling], page 473.

Sometimes, you may need to edit files containing many long logical lines, and it may not be practical to break them all up by adding newlines. In that case, you can use Visual Line mode, which enables *word wrapping*: instead of wrapping long lines exactly at the right window edge, Emacs wraps them at the word boundaries (i.e., space or tab characters) nearest to the right window edge. Visual Line mode also redefines editing commands such as `C-a`, `C-n`, and `C-k` to operate on screen lines rather than logical lines. See ⟨undefined⟩ [Visual Line Mode], page ⟨undefined⟩.

## 4.9 Cursor Position Information

Here are commands to get information about the size and position of parts of the buffer, and to count words and lines.

`M-x what-line`
: Display the line number of point.

`M-x line-number-mode`
`M-x column-number-mode`
: Toggle automatic display of the current line number or column number. See ⟨undefined⟩ [Optional Mode Line], page ⟨undefined⟩.

`M-=`
: Display the number of lines, words, and characters that are present in the region (`count-words-region`). See Chapter 8 [Mark], page 89, for information about the region.

`M-x count-words`
: Display the number of lines, words, and characters that are present in the buffer. If the region is active (see Chapter 8 [Mark], page 89), display the numbers for the region instead.

`C-x =`
: Display the character code of character after point, character position of point, and column of point (`what-cursor-position`).

`M-x hl-line-mode`
: Enable or disable highlighting of the current line. See ⟨undefined⟩ [Cursor Display], page ⟨undefined⟩.

`M-x size-indication-mode`
: Toggle automatic display of the size of the buffer. See ⟨undefined⟩ [Optional Mode Line], page ⟨undefined⟩.

`M-x what-line` displays the current line number in the echo area. This command is usually redundant, because the current line number is shown in the mode line (see Section 1.3 [Mode Line], page 8). However, if you narrow the buffer, the mode line shows the line

number relative to the accessible portion (see ⟨undefined⟩ [Narrowing], page ⟨undefined⟩). By contrast, `what-line` displays both the line number relative to the narrowed region and the line number relative to the whole buffer.

`M-=` (`count-words-region`) displays a message reporting the number of lines, words, and characters in the region (see Chapter 8 [Mark], page 89, for an explanation of the region). With a prefix argument, `C-u M-=`, the command displays a count for the entire buffer.

The command `M-x count-words` does the same job, but with a different calling convention. It displays a count for the region if the region is active, and for the buffer otherwise.

The command `C-x =` (`what-cursor-position`) shows information about the current cursor position and the buffer contents at that position. It displays a line in the echo area that looks like this:

```
Char: c (99, #o143, #x63) point=28062 of 36168 (78%) column=53
```

After '`Char:`', this shows the character in the buffer at point. The text inside the parenthesis shows the corresponding decimal, octal and hex character codes; for more information about how `C-x =` displays character information, see Section 19.1 [International Chars], page 374. After '`point=`' is the position of point as a character count (the first character in the buffer is position 1, the second character is position 2, and so on). The number after that is the total number of characters in the buffer, and the number in parenthesis expresses the position as a percentage of the total. After '`column=`' is the horizontal position of point, in columns counting from the left edge of the window.

If the buffer has been narrowed, making some of the text at the beginning and the end temporarily inaccessible, `C-x =` displays additional text describing the currently accessible range. For example, it might display this:

```
Char: C (67, #o103, #x43) point=252 of 889 (28%) <231-599> column=0
```

where the two extra numbers give the smallest and largest character position that point is allowed to assume. The characters between those two positions are the accessible ones. See ⟨undefined⟩ [Narrowing], page ⟨undefined⟩.

## 4.10 Numeric Arguments

In the terminology of mathematics and computing, *argument* means "data provided to a function or operation". You can give any Emacs command a *numeric argument* (also called a *prefix argument*). Some commands interpret the argument as a repetition count. For example, giving `C-f` an argument of ten causes it to move point forward by ten characters instead of one. With these commands, no argument is equivalent to an argument of one, and negative arguments cause them to move or act in the opposite direction.

The easiest way to specify a numeric argument is to type a digit and/or a minus sign while holding down the META key. For example,

```
M-5 C-n
```

moves down five lines. The keys `M-1`, `M-2`, and so on, as well as `M--`, are bound to commands (`digit-argument` and `negative-argument`) that set up an argument for the next command. `Meta--` without digits normally means −1.

If you enter more than one digit, you need not hold down the META key for the second and subsequent digits. Thus, to move down fifty lines, type

```
M-5 0 C-n
```

Note that this *does not* insert five copies of '0' and move down one line, as you might expect—the '0' is treated as part of the prefix argument.

(What if you do want to insert five copies of '0'? Type `M-5 C-u 0`. Here, `C-u` "terminates" the prefix argument, so that the next keystroke begins the command that you want to execute. Note that this meaning of `C-u` applies only to this case. For the usual role of `C-u`, see below.)

Instead of typing `M-1`, `M-2`, and so on, another way to specify a numeric argument is to type `C-u` (`universal-argument`) followed by some digits, or (for a negative argument) a minus sign followed by digits. A minus sign without digits normally means −1.

`C-u` alone has the special meaning of "four times": it multiplies the argument for the next command by four. `C-u C-u` multiplies it by sixteen. Thus, `C-u C-u C-f` moves forward sixteen characters. Other useful combinations are `C-u C-n`, `C-u C-u C-n` (move down a good fraction of a screen), `C-u C-u C-o` (make "a lot" of blank lines), and `C-u C-k` (kill four lines).

You can use a numeric argument before a self-inserting character to insert multiple copies of it. This is straightforward when the character is not a digit; for example, `C-u 6 4 a` inserts 64 copies of the character 'a'. But this does not work for inserting digits; `C-u 6 4 1` specifies an argument of 641. You can separate the argument from the digit to insert with another `C-u`; for example, `C-u 6 4 C-u 1` does insert 64 copies of the character '1'.

Some commands care whether there is an argument, but ignore its value. For example, the command `M-q` (`fill-paragraph`) fills text; with an argument, it justifies the text as well. (See Section 22.11 [Filling], page 473, for more information on `M-q`.) For these commands, it is enough to specify the argument with a single `C-u`.

Some commands use the value of the argument as a repeat count, but do something special when there is no argument. For example, the command `C-k` (`kill-line`) with argument *n* kills *n* lines, including their terminating newlines. But `C-k` with no argument is special: it kills the text up to the next newline, or, if point is right at the end of the line, it kills the newline itself. Thus, two `C-k` commands with no arguments can kill a nonblank line, just like `C-k` with an argument of one. (See Chapter 9 [Killing], page 95, for more information on `C-k`.)

A few commands treat a plain `C-u` differently from an ordinary argument. A few others may treat an argument of just a minus sign differently from an argument of −1. These unusual cases are described when they come up; they exist to make an individual command more convenient, and they are documented in that command's documentation string.

We use the term *prefix argument* to emphasize that you type such arguments before the command, and to distinguish them from minibuffer arguments (see Chapter 5 [Minibuffer], page 68), which are entered after invoking the command.

## 4.11 Repeating a Command

Many simple commands, such as those invoked with a single key or with `M-x` *`command-name`* `RET`, can be repeated by invoking them with a numeric argument that serves as a repeat count (see Section 4.10 [Arguments], page 65). However, if the command you want to repeat prompts for input, or uses a numeric argument in another way, that method won't work.

The command `C-x z` (`repeat`) provides another way to repeat an Emacs command many times. This command repeats the previous Emacs command, whatever that was. Repeating a command uses the same arguments that were used before; it does not read new arguments each time.

To repeat the command more than once, type additional `z`'s: each `z` repeats the command one more time. Repetition ends when you type a character other than `z`, or press a mouse button.

For example, suppose you type `C-u 2 0 C-d` to delete 20 characters. You can repeat that command (including its argument) three additional times, to delete a total of 80 characters, by typing `C-x z z z`. The first `C-x z` repeats the command once, and each subsequent `z` repeats it once again.

# 5 The Minibuffer

The *minibuffer* is where Emacs commands read complicated arguments, such as file names, buffer names, Emacs command names, or Lisp expressions. We call it the "minibuffer" because it's a special-purpose buffer with a small amount of screen space. You can use the usual Emacs editing commands in the minibuffer to edit the argument text.

## 5.1 Using the Minibuffer

When the minibuffer is in use, it appears in the echo area, with a cursor. The minibuffer starts with a *prompt*, usually ending with a colon. The prompt states what kind of input is expected, and how it will be used. The prompt is highlighted using the `minibuffer-prompt` face (see Section 11.12 [Faces], page 137).

The simplest way to enter a minibuffer argument is to type the text, then RET to submit the argument and exit the minibuffer. Alternatively, you can type `C-g` to exit the minibuffer by cancelling the command asking for the argument (see Section 34.1 [Quitting], page 717).

Sometimes, the prompt shows a *default argument*, inside parentheses before the colon. This default will be used as the argument if you just type RET. For example, commands that read buffer names usually show a buffer name as the default; you can type RET to operate on that default buffer.

If you enable Minibuffer Electric Default mode, a global minor mode, Emacs hides the default argument as soon as you modify the contents of the minibuffer (since typing RET would no longer submit that default). If you ever bring back the original minibuffer text, the prompt again shows the default. Furthermore, if you change the variable `minibuffer-eldef-shorten-default` to a non-`nil` value, the default argument is displayed as '`[default]`' instead of '`(default default)`', saving some screen space. To enable this minor mode, type `M-x minibuffer-electric-default-mode`.

Since the minibuffer appears in the echo area, it can conflict with other uses of the echo area. If an error message or an informative message is emitted while the minibuffer is active, the message hides the minibuffer for a few seconds, or until you type something; then the minibuffer comes back. While the minibuffer is in use, keystrokes do not echo.

## 5.2 Minibuffers for File Names

Commands such as `C-x C-f` (`find-file`) use the minibuffer to read a file name argument (see Section 4.5 [Basic Files], page 62). When the minibuffer is used to read a file name, it typically starts out with some initial text ending in a slash. This is the *default directory*. For example, it may start out like this:

```
Find file: /u2/emacs/src/
```

Here, '`Find file: `' is the prompt and '`/u2/emacs/src/`' is the default directory. If you now type `buffer.c` as input, that specifies the file '`/u2/emacs/src/buffer.c`'. See Section 15.8 [File Names], page 251, for information about the default directory.

You can specify the parent directory with '`..`': '`/a/b/../foo.el`' is equivalent to '`/a/foo.el`'. Alternatively, you can use `M-DEL` to kill directory names backwards (see ⟨undefined⟩ [Words], page ⟨undefined⟩).

To specify a file in a completely different directory, you can kill the entire default with `C-a C-k` (see Section 5.3 [Minibuffer Edit], page 69). Alternatively, you can ignore the default, and enter an absolute file name starting with a slash or a tilde after the default directory. For example, you can specify '`/etc/termcap`' as follows:

```
Find file: /u2/emacs/src//etc/termcap
```

Emacs interprets a double slash as "ignore everything before the second slash in the pair". In the example above, '`/u2/emacs/src/`' is ignored, so the argument you supplied is '`/etc/termcap`'. The ignored part of the file name is dimmed if the terminal allows it. (To disable this dimming, turn off File Name Shadow mode with the command `M-x file-name-shadow-mode`.)

Emacs interprets '`~/`' as your home directory. Thus, '`~/foo/bar.txt`' specifies a file named '`bar.txt`', inside a directory named '`foo`', which is in turn located in your home directory. In addition, '`~`*`user-id`*`/`' means the home directory of a user whose login name is *user-id*. Any leading directory name in front of the '`~`' is ignored: thus, '`/u2/emacs/~/foo/bar.txt`' is equivalent to '`~/foo/bar.txt`'.

On MS-Windows and MS-DOS systems, where a user doesn't always have a home directory, Emacs uses several alternatives. For MS-Windows, see Section I.5 [Windows HOME], page 784; for MS-DOS, see Section "MS-DOS File Names" in *the digital version of the Emacs Manual*. On these systems, the '`~`*`user-id`*`/`' construct is supported only for the current user, i.e., only if *user-id* is the current user's login name.

To prevent Emacs from inserting the default directory when reading file names, change the variable `insert-default-directory` to `nil`. In that case, the minibuffer starts out empty. Nonetheless, relative file name arguments are still interpreted based on the same default directory.

You can also enter remote file names in the minibuffer. See ⟨undefined⟩ [Remote Files], page ⟨undefined⟩.

## 5.3 Editing in the Minibuffer

The minibuffer is an Emacs buffer, albeit a peculiar one, and the usual Emacs commands are available for editing the argument text. (The prompt, however, is *read-only*, and cannot be changed.)

Since RET in the minibuffer submits the argument, you can't use it to insert a newline. You can do that with `C-q C-j`, which inserts a `C-j` control character, which is formally equivalent to a newline character (see Section 4.1 [Inserting Text], page 58). Alternatively, you can use the `C-o` (`open-line`) command (see Section 4.7 [Blank Lines], page 63).

Inside a minibuffer, the keys TAB, SPC, and `?` are often bound to *completion commands*, which allow you to easily fill in the desired text without typing all of it. See Section 5.4 [Completion], page 70. As with RET, you can use `C-q` to insert a TAB, SPC, or '`?`' character.

For convenience, `C-a` (`move-beginning-of-line`) in a minibuffer moves point to the beginning of the argument text, not the beginning of the prompt. For example, this allows you to erase the entire argument with `C-a C-k`.

When the minibuffer is active, the echo area is treated much like an ordinary Emacs window. For instance, you can switch to another window (with `C-x o`), edit text there, then return to the minibuffer window to finish the argument. You can even kill text in another

window, return to the minibuffer window, and yank the text into the argument. There are some restrictions on the minibuffer window, however: for instance, you cannot split it. See Chapter 17 [Windows], page 289.

Normally, the minibuffer window occupies a single screen line. However, if you add two or more lines' worth of text into the minibuffer, it expands automatically to accommodate the text. The variable `resize-mini-windows` controls the resizing of the minibuffer. The default value is `grow-only`, which means the behavior we have just described. If the value is `t`, the minibuffer window will also shrink automatically if you remove some lines of text from the minibuffer, down to a minimum of one screen line. If the value is `nil`, the minibuffer window never changes size automatically, but you can use the usual window-resizing commands on it (see Chapter 17 [Windows], page 289).

The variable `max-mini-window-height` controls the maximum height for resizing the minibuffer window. A floating-point number specifies a fraction of the frame's height; an integer specifies the maximum number of lines; `nil` means do not resize the minibuffer window automatically. The default value is 0.25.

The `C-M-v` command in the minibuffer scrolls the help text from commands that display help text of any sort in another window. You can also scroll the help text with `M-PRIOR` and `M-NEXT` (or, equivalently, `M-PAGEUP` and `M-PAGEDOWN`). This is especially useful with long lists of possible completions. See ⟨undefined⟩ [Other Window], page ⟨undefined⟩.

Emacs normally disallows most commands that use the minibuffer while the minibuffer is active. To allow such commands in the minibuffer, set the variable `enable-recursive-minibuffers` to `t`.

When not active, the minibuffer is in `minibuffer-inactive-mode`, and clicking `Mouse-1` there shows the '`*Messages*`' buffer. If you use a dedicated frame for minibuffers, Emacs also recognizes certain keys there, for example `n` to make a new frame.

## 5.4 Completion

You can often use a feature called *completion* to help enter arguments. This means that after you type part of the argument, Emacs can fill in the rest, or some of it, based on what was typed so far.

When completion is available, certain keys (usually TAB, RET, and SPC) are rebound in the minibuffer to special completion commands (see Section 5.4.2 [Completion Commands], page 71). These commands attempt to complete the text in the minibuffer, based on a set of *completion alternatives* provided by the command that requested the argument. You can usually type `?` to see a list of completion alternatives.

Although completion is usually done in the minibuffer, the feature is sometimes available in ordinary buffers too. See Section 23.8 [Symbol Completion], page 528.

### 5.4.1 Completion Example

A simple example may help here. `M-x` uses the minibuffer to read the name of a command, so completion works by matching the minibuffer text against the names of existing Emacs commands. Suppose you wish to run the command `auto-fill-mode`. You can do that by typing `M-x auto-fill-mode RET`, but it is easier to use completion.

If you type `M-x a u TAB`, the TAB looks for completion alternatives (in this case, command names) that start with '`au`'. There are several, including `auto-fill-mode` and

`autoconf-mode`, but they all begin with `auto`, so the '`au`' in the minibuffer completes to '`auto`'. (More commands may be defined in your Emacs session. For example, if a command called `authorize-me` was defined, Emacs could only complete as far as '`aut`'.)

If you type TAB again immediately, it cannot determine the next character; it could be '`-`', '`a`', or '`c`'. So it does not add any characters; instead, TAB displays a list of all possible completions in another window.

Next, type `-f`. The minibuffer now contains '`auto-f`', and the only command name that starts with this is `auto-fill-mode`. If you now type TAB, completion fills in the rest of the argument '`auto-fill-mode`' into the minibuffer.

Hence, typing just `a u TAB - f TAB` allows you to enter '`auto-fill-mode`'.

### 5.4.2 Completion Commands

Here is a list of the completion commands defined in the minibuffer when completion is allowed.

TAB
: Complete the text in the minibuffer as much as possible; if unable to complete, display a list of possible completions (`minibuffer-complete`).

SPC
: Complete up to one word from the minibuffer text before point (`minibuffer-complete-word`). This command is not available for arguments that often include spaces, such as file names.

RET
: Submit the text in the minibuffer as the argument, possibly completing first (`minibuffer-complete-and-exit`). See Section 5.4.3 [Completion Exit], page 72.

`?`
: Display a list of completions (`minibuffer-completion-help`).

TAB (`minibuffer-complete`) is the most fundamental completion command. It searches for all possible completions that match the existing minibuffer text, and attempts to complete as much as it can. See Section 5.4.4 [Completion Styles], page 72, for how completion alternatives are chosen.

SPC (`minibuffer-complete-word`) completes like TAB, but only up to the next hyphen or space. If you have '`auto-f`' in the minibuffer and type SPC, it finds that the completion is '`auto-fill-mode`', but it only inserts '`ill-`', giving '`auto-fill-`'. Another SPC at this point completes all the way to '`auto-fill-mode`'.

If TAB or SPC is unable to complete, it displays a list of matching completion alternatives (if there are any) in another window. You can display the same list with `?` (`minibuffer-completion-help`). The following commands can be used with the completion list:

`Mouse-1`
`Mouse-2`
: Clicking mouse button 1 or 2 on a completion alternative chooses it (`mouse-choose-completion`).

`M-v`
PAGEUP
PRIOR
: Typing `M-v`, while in the minibuffer, selects the window showing the completion list (`switch-to-completions`). This paves the way for using the commands below. PAGEUP or PRIOR does the same. You can also select the window in other ways (see Chapter 17 [Windows], page 289).

RET While in the completion list buffer, this chooses the completion at point (`choose-completion`).

RIGHT While in the completion list buffer, this moves point to the following completion alternative (`next-completion`).

LEFT While in the completion list buffer, this moves point to the previous completion alternative (`previous-completion`).

### 5.4.3 Completion Exit

When a command reads an argument using the minibuffer with completion, it also controls what happens when you type RET (`minibuffer-complete-and-exit`) to submit the argument. There are four types of behavior:

- *Strict completion* accepts only exact completion matches. Typing RET exits the minibuffer only if the minibuffer text is an exact match, or completes to one. Otherwise, Emacs refuses to exit the minibuffer; instead it tries to complete, and if no completion can be done it momentarily displays '`[No match]`' after the minibuffer text. (You can still leave the minibuffer by typing `C-g` to cancel the command.)

  An example of a command that uses this behavior is `M-x`, since it is meaningless for it to accept a non-existent command name.

- *Cautious completion* is like strict completion, except RET exits only if the text is already an exact match. If the text completes to an exact match, RET performs that completion but does not exit yet; you must type a second RET to exit.

  Cautious completion is used for reading file names for files that must already exist, for example.

- *Permissive completion* allows any input; the completion candidates are just suggestions. Typing RET does not complete, it just submits the argument as you have entered it.

- *Permissive completion with confirmation* is like permissive completion, with an exception: if you typed TAB and this completed the text up to some intermediate state (i.e., one that is not yet an exact completion match), typing RET right afterward does not submit the argument. Instead, Emacs asks for confirmation by momentarily displaying '`[Confirm]`' after the text; type RET again to confirm and submit the text. This catches a common mistake, in which one types RET before realizing that TAB did not complete as far as desired.

  You can tweak the confirmation behavior by customizing the variable `confirm-nonexistent-file-or-buffer`. The default value, `after-completion`, gives the behavior we have just described. If you change it to `nil`, Emacs does not ask for confirmation, falling back on permissive completion. If you change it to any other non-`nil` value, Emacs asks for confirmation whether or not the preceding command was TAB.

  This behavior is used by most commands that read file names, like `C-x C-f`, and commands that read buffer names, like `C-x b`.

### 5.4.4 How Completion Alternatives Are Chosen

Completion commands work by narrowing a large list of possible completion alternatives to a smaller subset that "matches" what you have typed in the minibuffer. In Section 5.4.1

[Completion Example], page 70, we gave a simple example of such matching. The procedure of determining what constitutes a "match" is quite intricate. Emacs attempts to offer plausible completions under most circumstances.

Emacs performs completion using one or more *completion styles*—sets of criteria for matching minibuffer text to completion alternatives. During completion, Emacs tries each completion style in turn. If a style yields one or more matches, that is used as the list of completion alternatives. If a style produces no matches, Emacs falls back on the next style.

The list variable `completion-styles` specifies the completion styles to use. Each list element is the name of a completion style (a Lisp symbol). The default completion styles are (in order):

`basic` A matching completion alternative must have the same beginning as the text in the minibuffer before point. Furthermore, if there is any text in the minibuffer after point, the rest of the completion alternative must contain that text as a substring.

`partial-completion`
This aggressive completion style divides the minibuffer text into words separated by hyphens or spaces, and completes each word separately. (For example, when completing command names, '`em-l-m`' completes to '`emacs-lisp-mode`'.)

Furthermore, a '`*`' in the minibuffer text is treated as a *wildcard*—it matches any character at the corresponding position in the completion alternative.

`emacs22` This completion style is similar to `basic`, except that it ignores the text in the minibuffer after point. It is so-named because it corresponds to the completion behavior in Emacs 22.

The following additional completion styles are also defined, and you can add them to `completion-styles` if you wish (see Chapter 33 [Customization], page 686):

`substring`
A matching completion alternative must contain the text in the minibuffer before point, and the text in the minibuffer after point, as substrings (in that same order).

Thus, if the text in the minibuffer is '`foobar`', with point between '`foo`' and '`bar`', that matches '`afoo`*b*`bar`*c*', where *a*, *b*, and *c* can be any string including the empty string.

`initials` This very aggressive completion style attempts to complete acronyms and initialisms. For example, when completing command names, it matches '`lch`' to '`list-command-history`'.

There is also a very simple completion style called `emacs21`. In this style, if the text in the minibuffer is '`foobar`', only matches starting with '`foobar`' are considered.

You can use different completion styles in different situations, by setting the variable `completion-category-overrides`. For example, the default setting says to use only `basic` and `substring` completion for buffer names.

### 5.4.5 Completion Options

Case is significant when completing case-sensitive arguments, such as command names. For example, when completing command names, '`AU`' does not complete to '`auto-fill-mode`'. Case differences are ignored when completing arguments in which case does not matter.

When completing file names, case differences are ignored if the variable `read-file-name-completion-ignore-case` is non-`nil`. The default value is `nil` on systems that have case-sensitive file-names, such as GNU/Linux; it is non-`nil` on systems that have case-insensitive file-names, such as Microsoft Windows. When completing buffer names, case differences are ignored if the variable `read-buffer-completion-ignore-case` is non-`nil`; the default is `nil`.

When completing file names, Emacs usually omits certain alternatives that are considered unlikely to be chosen, as determined by the list variable `completion-ignored-extensions`. Each element in the list should be a string; any file name ending in such a string is ignored as a completion alternative. Any element ending in a slash ('`/`') represents a subdirectory name. The standard value of `completion-ignored-extensions` has several elements including `".o"`, `".elc"`, and `"~"`. For example, if a directory contains '`foo.c`' and '`foo.elc`', '`foo`' completes to '`foo.c`'. However, if *all* possible completions end in "ignored" strings, they are not ignored: in the previous example, '`foo.e`' completes to '`foo.elc`'. Emacs disregards `completion-ignored-extensions` when showing completion alternatives in the completion list.

If `completion-auto-help` is set to `nil`, the completion commands never display the completion list buffer; you must type `?` to display the list. If the value is `lazy`, Emacs only shows the completion list buffer on the second attempt to complete. In other words, if there is nothing to complete, the first TAB echoes '`Next char not unique`'; the second TAB shows the completion list buffer.

If `completion-cycle-threshold` is non-`nil`, completion commands can "cycle" through completion alternatives. Normally, if there is more than one completion alternative for the text in the minibuffer, a completion command completes up to the longest common substring. If you change `completion-cycle-threshold` to `t`, the completion command instead completes to the first of those completion alternatives; each subsequent invocation of the completion command replaces that with the next completion alternative, in a cyclic manner. If you give `completion-cycle-threshold` a numeric value *n*, completion commands switch to this cycling behavior only when there are *n* or fewer alternatives.

Icomplete mode presents a constantly-updated display that tells you what completions are available for the text you've entered so far. The command to enable or disable this minor mode is `M-x icomplete-mode`.

## 5.5 Minibuffer History

Every argument that you enter with the minibuffer is saved in a *minibuffer history list* so you can easily use it again later. You can use the following arguments to quickly fetch an earlier argument into the minibuffer:

`M-p`
UP
: Move to the previous item in the minibuffer history, an earlier argument (`previous-history-element`).

`M-n`
DOWN Move to the next item in the minibuffer history (`next-history-element`).

`M-r` *regexp* `RET`
Move to an earlier item in the minibuffer history that matches *regexp* (`previous-matching-history-element`).

`M-s` *regexp* `RET`
Move to a later item in the minibuffer history that matches *regexp* (`next-matching-history-element`).

While in the minibuffer, `M-p` or UP (`previous-history-element`) moves through the minibuffer history list, one item at a time. Each `M-p` fetches an earlier item from the history list into the minibuffer, replacing its existing contents. Typing `M-n` or DOWN (`next-history-element`) moves through the minibuffer history list in the opposite direction, fetching later entries into the minibuffer.

If you type `M-n` in the minibuffer when there are no later entries in the minibuffer history (e.g., if you haven't previously typed `M-p`), Emacs tries fetching from a list of default arguments: values that you are likely to enter. You can think of this as moving through the "future history" list.

If you edit the text inserted by the `M-p` or M-N minibuffer history commands, this does not change its entry in the history list. However, the edited argument does go at the end of the history list when you submit it.

You can use `M-r` (`previous-matching-history-element`) to search through older elements in the history list, and `M-s` (`next-matching-history-element`) to search through newer entries. Each of these commands asks for a *regular expression* as an argument, and fetches the first matching entry into the minibuffer. See Section 12.6 [Regexps], page 206, for an explanation of regular expressions. A numeric prefix argument *n* means to fetch the *n*th matching entry. These commands are unusual, in that they use the minibuffer to read the regular expression argument, even though they are invoked from the minibuffer. An upper-case letter in the regular expression makes the search case-sensitive (see Section 12.9 [Search Case], page 211).

You can also search through the history using an incremental search. See Section 12.1.7 [Isearch Minibuffer], page 203.

Emacs keeps separate history lists for several different kinds of arguments. For example, there is a list for file names, used by all the commands that read file names. Other history lists include buffer names, command names (used by `M-x`), and command arguments (used by commands like `query-replace`).

The variable `history-length` specifies the maximum length of a minibuffer history list; adding a new element deletes the oldest element if the list gets too long. If the value is `t`, there is no maximum length.

The variable `history-delete-duplicates` specifies whether to delete duplicates in history. If it is non-`nil`, adding a new element deletes from the list all other elements that are equal to it. The default is `nil`.

## 5.6 Repeating Minibuffer Commands

Every command that uses the minibuffer once is recorded on a special history list, the *command history*, together with the values of its arguments, so that you can repeat the entire command. In particular, every use of `M-x` is recorded there, since `M-x` uses the minibuffer to read the command name.

`C-x ESC ESC`
: Re-execute a recent minibuffer command from the command history (`repeat-complex-command`).

`M-x list-command-history`
: Display the entire command history, showing all the commands `C-x ESC ESC` can repeat, most recent first.

`C-x ESC ESC` re-executes a recent command that used the minibuffer. With no argument, it repeats the last such command. A numeric argument specifies which command to repeat; 1 means the last one, 2 the previous, and so on.

`C-x ESC ESC` works by turning the previous command into a Lisp expression and then entering a minibuffer initialized with the text for that expression. Even if you don't know Lisp, it will probably be obvious which command is displayed for repetition. If you type just RET, that repeats the command unchanged. You can also change the command by editing the Lisp expression before you execute it. The executed command is added to the front of the command history unless it is identical to the most recent item.

Once inside the minibuffer for `C-x ESC ESC`, you can use the usual minibuffer history commands (see Section 5.5 [Minibuffer History], page 74) to move through the history list. After finding the desired previous command, you can edit its expression as usual and then execute it by typing RET.

Incremental search does not, strictly speaking, use the minibuffer. Therefore, although it behaves like a complex command, it normally does not appear in the history list for `C-x ESC ESC`. You can make incremental search commands appear in the history by setting `isearch-resume-in-command-history` to a non-`nil` value. See Section 12.1 [Incremental Search], page 199.

The list of previous minibuffer-using commands is stored as a Lisp list in the variable `command-history`. Each element is a Lisp expression that describes one command and its arguments. Lisp programs can re-execute a command by calling `eval` with the `command-history` element.

## 5.7 Entering passwords

Sometimes, you may need to enter a password into Emacs. For instance, when you tell Emacs to visit a file on another machine via a network protocol such as FTP, you often need to supply a password to gain access to the machine (see ⟨undefined⟩ [Remote Files], page ⟨undefined⟩).

Entering a password is similar to using a minibuffer. Emacs displays a prompt in the echo area (such as '`Password: `'); after you type the required password, press RET to submit it. To prevent others from seeing your password, every character you type is displayed as a dot ('`.`') instead of its usual form.

Most of the features and commands associated with the minibuffer can *not* be used when entering a password. There is no history or completion, and you cannot change windows or perform any other action with Emacs until you have submitted the password.

While you are typing the password, you may press DEL to delete backwards, removing the last character entered. `C-u` deletes everything you have typed so far. `C-g` quits the password prompt (see Section 34.1 [Quitting], page 717). `C-y` inserts the current kill into the password (see Chapter 9 [Killing], page 95). You may type either RET or ESC to submit the password. Any other self-inserting character key inserts the associated character into the password, and all other input is ignored.

## 5.8 Yes or No Prompts

An Emacs command may require you to answer a "yes or no" question during the course of its execution. Such queries come in two main varieties.

For the first type of "yes or no" query, the prompt ends with '`(y or n)`'. Such a query does not actually use the minibuffer; the prompt appears in the echo area, and you answer by typing either '`y`' or '`n`', which immediately delivers the response. For example, if you type `C-x C-w` (`write-file`) to save a buffer, and enter the name of an existing file, Emacs issues a prompt like this:

```
File 'foo.el' exists; overwrite? (y or n)
```

Because this query does not actually use the minibuffer, the usual minibuffer editing commands cannot be used. However, you can perform some window scrolling operations while the query is active: `C-l` recenters the selected window; `M-v` (or PAGEDOWN or NEXT) scrolls forward; `C-v` (or PAGEUP, or PRIOR) scrolls backward; `C-M-v` scrolls forward in the next window; and `C-M-S-v` scrolls backward in the next window. Typing `C-g` dismisses the query, and quits the command that issued it (see Section 34.1 [Quitting], page 717).

The second type of "yes or no" query is typically employed if giving the wrong answer would have serious consequences; it uses the minibuffer, and features a prompt ending with '`(yes or no)`'. For example, if you invoke `C-x k` (`kill-buffer`) on a file-visiting buffer with unsaved changes, Emacs activates the minibuffer with a prompt like this:

```
Buffer foo.el modified; kill anyway? (yes or no)
```

To answer, you must type '`yes`' or '`no`' into the minibuffer, followed by RET. The minibuffer behaves as described in the previous sections; you can switch to another window with `C-x o`, use the history commands `M-p` and `M-f`, etc. Type `C-g` to quit the minibuffer and the querying command.

# 6 Running Commands by Name

Every Emacs command has a name that you can use to run it. For convenience, many commands also have key bindings. You can run those commands by typing the keys, or run them by name. Most Emacs commands have no key bindings, so the only way to run them is by name. (See Section 33.3 [Key Bindings], page 702, for how to set up key bindings.)

By convention, a command name consists of one or more words, separated by hyphens; for example, `auto-fill-mode` or `manual-entry`. Command names mostly use complete English words to make them easier to remember.

To run a command by name, start with `M-x`, type the command name, then terminate it with RET. `M-x` uses the minibuffer to read the command name. The string '`M-x`' appears at the beginning of the minibuffer as a *prompt* to remind you to enter a command name to be run. RET exits the minibuffer and runs the command. See Chapter 5 [Minibuffer], page 68, for more information on the minibuffer.

You can use completion to enter the command name. For example, to invoke the command `forward-char`, you can type

```
M-x forward-char RET
```

or

```
M-x forw TAB c RET
```

Note that `forward-char` is the same command that you invoke with the key `C-f`. The existence of a key binding does not stop you from running the command by name.

To cancel the `M-x` and not run a command, type `C-g` instead of entering the command name. This takes you back to command level.

To pass a numeric argument to the command you are invoking with `M-x`, specify the numeric argument before `M-x`. The argument value appears in the prompt while the command name is being read, and finally `M-x` passes the argument to that command.

When the command you run with `M-x` has a key binding, Emacs mentions this in the echo area after running the command. For example, if you type `M-x forward-word`, the message says that you can run the same command by typing `M-f`. You can turn off these messages by setting the variable `suggest-key-bindings` to `nil`.

In this manual, when we speak of running a command by name, we often omit the RET that terminates the name. Thus we might say `M-x auto-fill-mode` rather than `M-x auto-fill-mode RET`. We mention the RET only for emphasis, such as when the command is followed by arguments.

`M-x` works by running the command `execute-extended-command`, which is responsible for reading the name of another command and invoking it.

# 7 Documentation

GNU Emacs has convenient built-in help facilities, most of which derive their information from documentation strings associated with functions and variables. This chapter describes how to access documentation strings in Lisp programs. See ⟨undefined⟩ [Documentation Tips], page ⟨undefined⟩, for how to write good documentation strings.

Note that the documentation strings for Emacs are not the same thing as the Emacs manual. Manuals have their own source files, written in the Texinfo language; documentation strings are specified in the definitions of the functions and variables they apply to. A collection of documentation strings is not sufficient as a manual because a good manual is not organized in that fashion; it is organized in terms of topics of discussion.

For commands to display documentation strings, see Section "Help" in *The GNU Emacs Manual*.

## 7.1 Documentation Basics

A documentation string is written using the Lisp syntax for strings, with double-quote characters surrounding the text of the string. This is because it really is a Lisp string object. The string serves as documentation when it is written in the proper place in the definition of a function or variable. In a function definition, the documentation string follows the argument list. In a variable definition, the documentation string follows the initial value of the variable.

When you write a documentation string, make the first line a complete sentence (or two complete sentences) that briefly describes what the function or variable does. Some commands, such as `apropos`, show only the first line of a multi-line documentation string. Also, you should not indent the second line of a documentation string, if it has one, because that looks odd when you use `C-h f` (`describe-function`) or `C-h v` (`describe-variable`) to view the documentation string. There are many other conventions for documentation strings; see ⟨undefined⟩ [Documentation Tips], page ⟨undefined⟩.

Documentation strings can contain several special text sequences, referring to key bindings which are looked up in the current keymaps when the user views the documentation. This allows the help commands to display the correct keys even if a user rearranges the default key bindings. See Section 7.3 [Keys in Documentation], page 82.

In the documentation string of an autoloaded command (see ⟨undefined⟩ [Autoload], page ⟨undefined⟩), these special text sequences have an additional special effect: they cause `C-h f` (`describe-function`) on the command to trigger autoloading. (This is needed for correctly setting up the hyperlinks in the '`*Help*`' buffer).

Emacs Lisp mode fills documentation strings to the width specified by `emacs-lisp-docstring-fill-column`.

Exactly where a documentation string is stored depends on how its function or variable was defined or loaded into memory:

- When you define a function (see ⟨undefined⟩ [Lambda Expressions], page ⟨undefined⟩, and see ⟨undefined⟩ [Function Documentation], page ⟨undefined⟩), the documentation string is stored in the function definition itself. You can also put function documentation in the `function-documentation` property of a function name. That is useful for function definitions which can't hold a documentation string, such as keyboard macros.

- When you define a variable with a `defvar` or related form (see ⟨undefined⟩ [Defining Variables], page ⟨undefined⟩), the documentation is stored in the variable's `variable-documentation` property.
- To save memory, the documentation for preloaded functions and variables (including primitive functions and autoloaded functions) is not kept in memory, but in the file '`emacs/etc/DOC-`*version*', where *version* is the Emacs version number (see ⟨undefined⟩ [Version Info], page ⟨undefined⟩).
- When a function or variable is loaded from a byte-compiled file during the Emacs session, its documentation string is not loaded into memory. Instead, Emacs looks it up in the byte-compiled file as needed. See ⟨undefined⟩ [Docs and Compilation], page ⟨undefined⟩.

Regardless of where the documentation string is stored, you can retrieve it using the `documentation` or `documentation-property` function, described in the next section.

## 7.2 Access to Documentation Strings

`documentation-property` *symbol property* **&optional** *verbatim* [Function]

This function returns the documentation string recorded in *symbol*'s property list under property *property*. It is most often used to look up the documentation strings of variables, for which *property* is `variable-documentation`. However, it can also be used to look up other kinds of documentation, such as for customization groups (but for function documentation, use the `documentation` command, below).

If the value recorded in the property list refers to a documentation string stored in a '`DOC-`*version*' file or a byte-compiled file, it looks up that string and returns it. If the property value isn't `nil`, isn't a string, and doesn't refer to text in a file, then it is evaluated as a Lisp expression to obtain a string.

The last thing this function does is pass the string through `substitute-command-keys` to substitute actual key bindings (see Section 7.3 [Keys in Documentation], page 82). However, it skips this step if *verbatim* is non-`nil`.

```
(documentation-property 'command-line-processed
   'variable-documentation)
     ⇒ "Non-nil once command line has been processed"
(symbol-plist 'command-line-processed)
     ⇒ (variable-documentation 188902)
(documentation-property 'emacs 'group-documentation)
     ⇒ "Customization of the One True Editor."
```

`documentation` *function* **&optional** *verbatim* [Function]

This function returns the documentation string of *function*. It handles macros, named keyboard macros, and special forms, as well as ordinary functions.

If *function* is a symbol, this function first looks for the `function-documentation` property of that symbol; if that has a non-`nil` value, the documentation comes from that value (if the value is not a string, it is evaluated). If *function* is not a symbol, or if it has no `function-documentation` property, then `documentation` extracts the documentation string from the actual function definition, reading it from a file if called for.

Finally, unless *verbatim* is non-`nil`, it calls `substitute-command-keys` so as to return a value containing the actual (current) key bindings.

The function `documentation` signals a `void-function` error if *function* has no function definition. However, it is OK if the function definition has no documentation string. In that case, `documentation` returns `nil`.

`face-documentation` *face* [Function]

This function returns the documentation string of *face* as a face.

Here is an example of using the two functions, `documentation` and `documentation-property`, to display the documentation strings for several symbols in a '`*Help*`' buffer.

```
(defun describe-symbols (pattern)
  "Describe the Emacs Lisp symbols matching PATTERN.
All symbols that have PATTERN in their name are described
in the `*Help*' buffer."
  (interactive "sDescribe symbols matching: ")
  (let ((describe-func
         (function
          (lambda (s)
            ;; Print description of symbol.
            (if (fboundp s)             ; It is a function.
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (commandp s)
                       (let ((keys (where-is-internal s)))
                         (if keys
                             (concat
                              "Keys: "
                              (mapconcat 'key-description
                                         keys " "))
                           "Keys: none"))
                     "Function")
                   (or (documentation s)
                       "not documented"))))

            (if (boundp s)              ; It is a variable.
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (custom-variable-p s)
                       "Option " "Variable")
                   (or (documentation-property
                        s 'variable-documentation)
                       "not documented")))))))
        sym-list)

    ;; Build a list of symbols that match pattern.
    (mapatoms (function
               (lambda (sym)
                 (if (string-match pattern (symbol-name sym))
                     (setq sym-list (cons sym sym-list))))))

    ;; Display the data.
    (help-setup-xref (list 'describe-symbols pattern) (interactive-p))
    (with-help-window (help-buffer)
      (mapcar describe-func (sort sym-list 'string<)))))
```

The `describe-symbols` function works like `apropos`, but provides more information.

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
Semipermanent goal column for vertical motion, as set by ...

set-goal-column Keys: C-x C-n
Set the current horizontal position as a goal for C-n and C-p.
Those commands will move to this position in the line moved to
rather than trying to keep the same horizontal position.
With a non-nil argument, clears out the goal column
so that C-n and C-p resume vertical motion.
The goal column is stored in the variable ‘goal-column’.

temporary-goal-column   Variable
Current goal column for vertical motion.
It is the column where point was
at the start of current run of vertical motion commands.
When the ‘track-eol’ feature is doing its job, the value is 9999.
---------- Buffer: *Help* ----------
```

`Snarf-documentation` *filename* [Function]

This function is used when building Emacs, just before the runnable Emacs is dumped. It finds the positions of the documentation strings stored in the file *filename*, and records those positions into memory in the function definitions and variable property lists. See ⟨undefined⟩ [Building Emacs], page ⟨undefined⟩.

Emacs reads the file *filename* from the ‘`emacs/etc`’ directory. When the dumped Emacs is later executed, the same file will be looked for in the directory `doc-directory`. Usually *filename* is `"DOC-`*`version`*`"`.

`doc-directory` [Variable]

This variable holds the name of the directory which should contain the file `"DOC-`*`version`*`"` that contains documentation strings for built-in and preloaded functions and variables.

In most cases, this is the same as `data-directory`. They may be different when you run Emacs from the directory where you built it, without actually installing it. See [Definition of data-directory], page 87.

## 7.3 Substituting Key Bindings in Documentation

When documentation strings refer to key sequences, they should use the current, actual key bindings. They can do so using certain special text sequences described below. Accessing documentation strings in the usual way substitutes current key binding information for these special sequences. This works by calling `substitute-command-keys`. You can also call that function yourself.

Here is a list of the special sequences and what they mean:

`\[`*`command`*`]`

stands for a key sequence that will invoke *command*, or ‘`M-x` *command*’ if *command* has no key bindings.

`\{`*mapvar*`}`
: stands for a summary of the keymap which is the value of the variable *mapvar*. The summary is made using `describe-bindings`.

`\<`*mapvar*`>`
: stands for no text itself. It is used only for a side effect: it specifies *mapvar*'s value as the keymap for any following '`\[`*command*`]`' sequences in this documentation string.

`\=`
: quotes the following character and is discarded; thus, '`\=\[`' puts '`\[`' into the output, and '`\=\=`' puts '`\=`' into the output.

**Please note:** Each '`\`' must be doubled when written in a string in Emacs Lisp.

`substitute-command-keys` *string* [Function]
: This function scans *string* for the above special sequences and replaces them by what they stand for, returning the result as a string. This permits display of documentation that refers accurately to the user's own customized key bindings.

  If a command has multiple bindings, this function normally uses the first one it finds. You can specify one particular key binding by assigning an `:advertised-binding` symbol property to the command, like this:

```
(put 'undo :advertised-binding [?\C-/])
```

  The `:advertised-binding` property also affects the binding shown in menu items (see Section 1.4 [Menu Bar], page 9). The property is ignored if it specifies a key binding that the command does not actually have.

Here are examples of the special sequences:

```
(substitute-command-keys
   "To abort recursive edit, type: \\[abort-recursive-edit]")
⇒ "To abort recursive edit, type: C-]"

(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
⇒ "The keys that are defined for the minibuffer here are:

?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
C-j             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

(substitute-command-keys
   "To abort a recursive edit from the minibuffer, type\
\\<minibuffer-local-must-match-map>\\[abort-recursive-edit].")
⇒ "To abort a recursive edit from the minibuffer, type C-g."
```

There are other special conventions for the text in documentation strings—for instance, you can refer to functions, variables, and sections of this manual. See ⟨undefined⟩ [Documentation Tips], page ⟨undefined⟩, for details.

## 7.4 Describing Characters for Help Messages

These functions convert events, key sequences, or characters to textual descriptions. These descriptions are useful for including arbitrary text characters or key sequences in messages, because they convert non-printing and whitespace characters to sequences of printing characters. The description of a non-whitespace printing character is the character itself.

`key-description` *sequence* **&optional** *prefix* [Function]

This function returns a string containing the Emacs standard notation for the input events in *sequence*. If *prefix* is non-`nil`, it is a sequence of input events leading up to *sequence* and is included in the return value. Both arguments may be strings, vectors or lists. See Section 2.7 [Input Events], page 23, for more information about valid events.

```
(key-description [?\M-3 delete])
     ⇒ "M-3 <delete>"
(key-description [delete] "\M-3")
     ⇒ "M-3 <delete>"
```

See also the examples for `single-key-description`, below.

`single-key-description` *event* **&optional** *no-angles* [Function]

This function returns a string describing *event* in the standard Emacs notation for keyboard input. A normal printing character appears as itself, but a control character turns into a string starting with '`C-`', a meta character turns into a string starting with '`M-`', and space, tab, etc., appear as '`SPC`', '`TAB`', etc. A function key symbol appears inside angle brackets '`<...>`'. An event that is a list appears as the name of the symbol in the CAR of the list, inside angle brackets.

If the optional argument *no-angles* is non-`nil`, the angle brackets around function keys and event symbols are omitted; this is for compatibility with old versions of Emacs which didn't use the brackets.

```
(single-key-description ?\C-x)
     ⇒ "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     ⇒ "C-x SPC M-y SPC C-j SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'delete)
     ⇒ "<delete>"
(single-key-description 'C-mouse-1)
     ⇒ "<C-mouse-1>"
(single-key-description 'C-mouse-1 t)
     ⇒ "C-mouse-1"
```

`text-char-description` *character* [Function]

This function returns a string describing *character* in the standard Emacs notation for characters that appear in text—like `single-key-description`, except that control characters are represented with a leading caret (which is how control characters in Emacs buffers are usually displayed). Another difference is that `text-char-description` recognizes the 2**7 bit as the Meta character, whereas `single-key-description` uses the 2**27 bit for Meta.

```
(text-char-description ?\C-c)
     ⇒ "^C"
(text-char-description ?\M-m)
     ⇒ "\xed"
```

```
(text-char-description ?\C-\M-m)
     ⇒ "\x8d"
(text-char-description (+ 128 ?m))
     ⇒ "M-m"
(text-char-description (+ 128 ?\C-m))
     ⇒ "M-^M"
```

`read-kbd-macro` *string* **&optional** *need-vector* [Command]

This function is used mainly for operating on keyboard macros, but it can also be used as a rough inverse for `key-description`. You call it with a string containing key descriptions, separated by spaces; it returns a string or vector containing the corresponding events. (This may or may not be a single valid key sequence, depending on what events you use; see ⟨undefined⟩ [Key Sequences], page ⟨undefined⟩.) If *need-vector* is non-`nil`, the return value is always a vector.

## 7.5 Help Functions

Emacs provides a variety of on-line help functions, all accessible to the user as subcommands of the prefix `C-h`. For more information about them, see Section "Help" in *The GNU Emacs Manual*. Here we describe some program-level interfaces to the same information.

`apropos` *pattern* **&optional** *do-all* [Command]

This function finds all "meaningful" symbols whose names contain a match for the apropos pattern *pattern*. An apropos pattern is either a word to match, a space-separated list of words of which at least two must match, or a regular expression (if any special regular expression characters occur). A symbol is "meaningful" if it has a definition as a function, variable, or face, or has properties.

The function returns a list of elements that look like this:

```
(symbol score function-doc variable-doc
 plist-doc widget-doc face-doc group-doc)
```

Here, *score* is an integer measure of how important the symbol seems to be as a match. Each of the remaining elements is a documentation string, or `nil`, for *symbol* as a function, variable, etc.

It also displays the symbols in a buffer named '`*Apropos*`', each with a one-line description taken from the beginning of its documentation string.

If *do-all* is non-`nil`, or if the user option `apropos-do-all` is non-`nil`, then `apropos` also shows key bindings for the functions that are found; it also shows *all* interned symbols, not just meaningful ones (and it lists them in the return value as well).

`help-map` [Variable]

The value of this variable is a local keymap for characters following the Help key, `C-h`.

`help-command` [Prefix Command]

This symbol is not a function; its function definition cell holds the keymap known as `help-map`. It is defined in '`help.el`' as follows:

```
(define-key global-map (string help-char) 'help-command)
(fset 'help-command help-map)
```

`help-char` [User Option]

The value of this variable is the help character—the character that Emacs recognizes as meaning Help. By default, its value is 8, which stands for `C-h`. When Emacs reads this character, if `help-form` is a non-`nil` Lisp expression, it evaluates that expression, and displays the result in a window if it is a string.

Usually the value of `help-form` is `nil`. Then the help character has no special meaning at the level of command input, and it becomes part of a key sequence in the normal way. The standard key binding of `C-h` is a prefix key for several general-purpose help features.

The help character is special after prefix keys, too. If it has no binding as a subcommand of the prefix key, it runs `describe-prefix-bindings`, which displays a list of all the subcommands of the prefix key.

`help-event-list` [User Option]

The value of this variable is a list of event types that serve as alternative "help characters". These events are handled just like the event specified by `help-char`.

`help-form` [Variable]

If this variable is non-`nil`, its value is a form to evaluate whenever the character `help-char` is read. If evaluating the form produces a string, that string is displayed.

A command that calls `read-event`, `read-char-choice`, or `read-char` probably should bind `help-form` to a non-`nil` expression while it does input. (The time when you should not do this is when `C-h` has some other meaning.) Evaluating this expression should result in a string that explains what the input is for and how to enter it properly.

Entry to the minibuffer binds this variable to the value of `minibuffer-help-form` (see ⟨undefined⟩ [Definition of minibuffer-help-form], page ⟨undefined⟩).

`prefix-help-command` [Variable]

This variable holds a function to print help for a prefix key. The function is called when the user types a prefix key followed by the help character, and the help character has no binding after that prefix. The variable's default value is `describe-prefix-bindings`.

`describe-prefix-bindings` [Command]

This function calls `describe-bindings` to display a list of all the subcommands of the prefix key of the most recent key sequence. The prefix described consists of all but the last event of that key sequence. (The last event is, presumably, the help character.)

The following two functions are meant for modes that want to provide help without relinquishing control, such as the "electric" modes. Their names begin with '`Helper`' to distinguish them from the ordinary help functions.

`Helper-describe-bindings` [Command]

This command pops up a window displaying a help buffer containing a listing of all of the key bindings from both the local and global keymaps. It works by calling `describe-bindings`.

`Helper-help` [Command]
: This command provides help for the current mode. It prompts the user in the minibuffer with the message '`Help (Type ? for further options)`', and then provides assistance in finding out what the key bindings are, and what the mode is intended for. It returns `nil`.

  This can be customized by changing the map `Helper-help-map`.

`data-directory` [Variable]
: This variable holds the name of the directory in which Emacs finds certain documentation and text files that come with Emacs.

`help-buffer` [Function]
: This function returns the name of the help buffer, which is normally '`*Help*`'; if such a buffer does not exist, it is first created.

`with-help-window` *buffer-name body...* [Macro]
: This macro evaluates the *body* forms, inserting any output they produce into a buffer named *buffer-name* like `with-output-to-temp-buffer` (see Section 11.8 [Temporary Displays], page 125). (Usually, *buffer-name* should be the value returned by the function `help-buffer`.) It also puts the specified buffer into Help mode and displays a message telling the user how to quit and scroll the help window.

`help-setup-xref` *item interactive-p* [Function]
: This function updates the cross reference data in the '`*Help*`' buffer, which is used to regenerate the help information when the user clicks on the '`Back`' or '`Forward`' buttons. Most commands that use the '`*Help*`' buffer should invoke this function before clearing the buffer. The *item* argument should have the form `(`*`function . args`*`)`, where *function* is a function to call, with argument list *args*, to regenerate the help buffer. The *interactive-p* argument is non-`nil` if the calling command was invoked interactively; in that case, the stack of items for the '`*Help*`' buffer's '`Back`' buttons is cleared.

See [describe-symbols example], page 81, for an example of using `help-buffer`, `with-help-window`, and `help-setup-xref`.

`make-help-screen` *fname help-line help-text help-map* [Macro]
: This macro defines a help command named *fname* that acts like a prefix key that shows a list of the subcommands it offers.

  When invoked, *fname* displays *help-text* in a window, then reads and executes a key sequence according to *help-map*. The string *help-text* should describe the bindings available in *help-map*.

  The command *fname* is defined to handle a few events itself, by scrolling the display of *help-text*. When *fname* reads one of those special events, it does the scrolling and then reads another event. When it reads an event that is not one of those few, and which has a binding in *help-map*, it executes that key's binding and then returns.

  The argument *help-line* should be a single-line summary of the alternatives in *help-map*. In the current version of Emacs, this argument is used only if you set the option `three-step-help` to `t`.

  This macro is used in the command `help-for-help` which is the binding of `C-h C-h`.

`three-step-help` [User Option]

If this variable is non-`nil`, commands defined with `make-help-screen` display their *help-line* strings in the echo area at first, and display the longer *help-text* strings only if the user types the help character again.

# 8 The Mark and the Region

Many Emacs commands operate on an arbitrary contiguous part of the current buffer. To specify the text for such a command to operate on, you set *the mark* at one end of it, and move point to the other end. The text between point and the mark is called *the region*. The region always extends between point and the mark, no matter which one comes earlier in the text; each time you move point, the region changes.

Setting the mark at a position in the text also *activates* it. When the mark is active, we say also that the region is active; Emacs indicates its extent by highlighting the text within it, using the `region` face (see Section 33.1.5 [Face Customization], page 690).

After certain non-motion commands, including any command that changes the text in the buffer, Emacs automatically *deactivates* the mark; this turns off the highlighting. You can also explicitly deactivate the mark at any time, by typing `C-g` (see Section 34.1 [Quitting], page 717).

The above default behavior is known as Transient Mark mode. Disabling Transient Mark mode switches Emacs to an alternative behavior, in which the region is usually not highlighted. See Section 8.7 [Disabled Transient Mark], page 94.

Setting the mark in one buffer has no effect on the marks in other buffers. When you return to a buffer with an active mark, the mark is at the same place as before. When multiple windows show the same buffer, they can have different values of point, and thus different regions, but they all share one common mark position. See Chapter 17 [Windows], page 289. Ordinarily, only the selected window highlights its region; however, if the variable `highlight-nonselected-windows` is non-`nil`, each window highlights its own region.

## 8.1 Setting the Mark

Here are some commands for setting the mark:

`C-SPC` Set the mark at point, and activate it (`set-mark-command`).

`C-@` The same.

`C-x C-x` Set the mark at point, and activate it; then move point where the mark used to be (`exchange-point-and-mark`).

`Drag-Mouse-1`
Set point and the mark around the text you drag across.

`Mouse-3` Set the mark at point, then move point to where you click (`mouse-save-then-kill`).

`'Shifted cursor motion keys'`
Set the mark at point if the mark is inactive, then move point. See Section 8.6 [Shift Selection], page 93.

The most common way to set the mark is with `C-SPC` (`set-mark-command`)[1]. This sets the mark where point is, and activates it. You can then move point away, leaving the mark behind.

[1] There is no `C-SPC` character in ASCII; usually, typing `C-SPC` on a text terminal gives the character `C-@`. This key is also bound to `set-mark-command`, so unless you are unlucky enough to have a text terminal that behaves differently, you might as well think of `C-@` as `C-SPC`.

For example, suppose you wish to convert part of the buffer to upper case. To accomplish this, go to one end of the desired text, type `C-SPC`, and move point until the desired portion of text is highlighted. Now type `C-x C-u` (`upcase-region`). This converts the text in the region to upper case, and then deactivates the mark.

Whenever the mark is active, you can deactivate it by typing `C-g` (see Section 34.1 [Quitting], page 717). Most commands that operate on the region also automatically deactivate the mark, like `C-x C-u` in the above example.

Instead of setting the mark in order to operate on a region, you can also use it to "remember" a position in the buffer (by typing `C-SPC C-SPC`), and later jump back there (by typing `C-u C-SPC`). See Section 8.4 [Mark Ring], page 92, for details.

The command `C-x C-x` (`exchange-point-and-mark`) exchanges the positions of point and the mark. `C-x C-x` is useful when you are satisfied with the position of point but want to move the other end of the region (where the mark is). Using `C-x C-x` a second time, if necessary, puts the mark at the new position with point back at its original position. Normally, if the mark is inactive, this command first reactivates the mark wherever it was last set, to ensure that the region is left highlighted. However, if you call it with a prefix argument, it leaves the mark inactive and the region unhighlighted; you can use this to jump to the mark in a manner similar to `C-u C-SPC`.

You can also set the mark with the mouse. If you press the left mouse button (`down-mouse-1`) and drag the mouse across a range of text, this sets the mark where you first pressed the mouse button and puts point where you release it. Alternatively, clicking the right mouse button (`mouse-3`) sets the mark at point and then moves point to where you clicked. See ⟨undefined⟩ [Mouse Commands], page ⟨undefined⟩, for a more detailed description of these mouse commands.

Finally, you can set the mark by holding down the shift key while typing certain cursor motion commands (such as `S-RIGHT`, `S-C-f`, `S-C-n`, etc.). This is called *shift-selection*. It sets the mark at point before moving point, but only if there is no active mark set via shift-selection. The mark set by mouse commands and by shift-selection behaves slightly differently from the usual mark: any subsequent unshifted cursor motion command deactivates it automatically. For details, See Section 8.6 [Shift Selection], page 93.

Many commands that insert text, such as `C-y` (`yank`), set the mark at the other end of the inserted text, without activating it. This lets you easily return to that position (see Section 8.4 [Mark Ring], page 92). You can tell that a command does this when it shows '`Mark set`' in the echo area.

Under X, every time the active region changes, Emacs saves the text in the region to the *primary selection*. This lets you insert that text into other X applications with `mouse-2` clicks. See Section 9.3.2 [Primary Selection], page 101.

## 8.2 Commands to Mark Textual Objects

Here are commands for placing point and the mark around a textual object such as a word, list, paragraph or page:

`M-@` Set mark after end of next word (`mark-word`). This does not move point.

`C-M-@` Set mark after end of following balanced expression (`mark-sexp`). This does not move point.

`M-h` Move point to the beginning of the current paragraph, and set mark at the end (`mark-paragraph`).

`C-M-h` Move point to the beginning of the current defun, and set mark at the end (`mark-defun`).

`C-x C-p` Move point to the beginning of the current page, and set mark at the end (`mark-page`).

`C-x h` Move point to the beginning of the buffer, and set mark at the end (`mark-whole-buffer`).

`M-@` (`mark-word`) sets the mark at the end of the next word (see ⟨undefined⟩ [Words], page ⟨undefined⟩, for information about words). Repeated invocations of this command extend the region by advancing the mark one word at a time. As an exception, if the mark is active and located before point, `M-@` moves the mark backwards from its current position one word at a time.

This command also accepts a numeric argument *n*, which tells it to advance the mark by *n* words. A negative argument moves the mark back by *n* words.

Similarly, `C-M-@` (`mark-sexp`) puts the mark at the end of the next balanced expression (see Section 23.4.1 [Expressions], page 521). Repeated invocations extend the region to subsequent expressions, while positive or negative numeric arguments move the mark forward or backward by the specified number of expressions.

The other commands in the above list set both point and mark, so as to delimit an object in the buffer. `M-h` (`mark-paragraph`) marks paragraphs (see ⟨undefined⟩ [Paragraphs], page ⟨undefined⟩), `C-M-h` (`mark-defun`) marks top-level definitions (see Section 23.2.2 [Moving by Defuns], page 515), and `C-x C-p` (`mark-page`) marks pages (see ⟨undefined⟩ [Pages], page ⟨undefined⟩). Repeated invocations again play the same role, extending the region to consecutive objects; similarly, numeric arguments specify how many objects to move the mark by.

`C-x h` (`mark-whole-buffer`) sets up the entire buffer as the region, by putting point at the beginning and the mark at the end.

## 8.3 Operating on the Region

Once you have a region, here are some of the ways you can operate on it:

- Kill it with `C-w` (see Chapter 9 [Killing], page 95).
- Copy it to the kill ring with `M-w` (see Section 22.8.3 [Yanking], page 465).
- Convert case with `C-x C-l` or `C-x C-u` (see ⟨undefined⟩ [Case], page ⟨undefined⟩).
- Undo changes within it using `C-u C-/` (see Section 22.9 [Undo], page 469).
- Replace text within it using `M-%` (see Section 12.10.4 [Query Replace], page 213).
- Indent it with `C-x TAB` or `C-M-\` (see Section 22.17 [Indentation], page 483).
- Fill it as text with `M-x fill-region` (see Section 22.11 [Filling], page 473).
- Check the spelling of words within it with `M-$` (see Section 13.4 [Spelling], page 219).
- Evaluate it as Lisp code with `M-x eval-region` (see Section 24.9 [Lisp Eval], page 550).
- Save it in a register with `C-x r s` (see Section 22.21 [Registers], page 507).

- Save it in a buffer or a file (see Section 9.4 [Accumulating Text], page 102).

Some commands have a default behavior when the mark is inactive, but operate on the region if the mark is active. For example, `M-$` (`ispell-word`) normally checks the spelling of the word at point, but it checks the text in the region if the mark is active (see Section 13.4 [Spelling], page 219). Normally, such commands use their default behavior if the region is empty (i.e., if mark and point are at the same position). If you want them to operate on the empty region, change the variable `use-empty-active-region` to `t`.

As described in Section 4.3 [Erasing], page 61, the DEL (`backward-delete-char`) and DELETE (`delete-forward-char`) commands also act this way. If the mark is active, they delete the text in the region. (As an exception, if you supply a numeric argument *n*, where *n* is not one, these commands delete *n* characters regardless of whether the mark is active). If you change the variable `delete-active-region` to `nil`, then these commands don't act differently when the mark is active. If you change the value to `kill`, these commands *kill* the region instead of deleting it (see Chapter 9 [Killing], page 95).

Other commands always operate on the region, and have no default behavior. Such commands usually have the word `region` in their names, like `C-w` (`kill-region`) and `C-x C-u` (`upcase-region`). If the mark is inactive, they operate on the "inactive region"—that is, on the text between point and the position at which the mark was last set (see Section 8.4 [Mark Ring], page 92). To disable this behavior, change the variable `mark-even-if-inactive` to `nil`. Then these commands will instead signal an error if the mark is inactive.

By default, text insertion occurs normally even if the mark is active—for example, typing `a` inserts the character '`a`', then deactivates the mark. If you enable Delete Selection mode, a minor mode, then inserting text while the mark is active causes the text in the region to be deleted first. To toggle Delete Selection mode on or off, type `M-x delete-selection-mode`.

## 8.4 The Mark Ring

Each buffer remembers previous locations of the mark, in the *mark ring*. Commands that set the mark also push the old mark onto this ring. One of the uses of the mark ring is to remember spots that you may want to go back to.

`C-SPC C-SPC`
: Set the mark, pushing it onto the mark ring, without activating it.

`C-u C-SPC`
: Move point to where the mark was, and restore the mark from the ring of former marks.

The command `C-SPC C-SPC` is handy when you want to use the mark to remember a position to which you may wish to return. It pushes the current point onto the mark ring, without activating the mark (which would cause Emacs to highlight the region). This is actually two consecutive invocations of `C-SPC` (`set-mark-command`); the first `C-SPC` sets the mark, and the second `C-SPC` deactivates it. (When Transient Mark mode is off, `C-SPC C-SPC` instead activates Transient Mark mode temporarily; see Section 8.7 [Disabled Transient Mark], page 94.)

To return to a marked position, use `set-mark-command` with a prefix argument: `C-u C-SPC`. This moves point to where the mark was, and deactivates the mark if it was active.

Each subsequent `C-u C-SPC` jumps to a prior position stored in the mark ring. The positions you move through in this way are not lost; they go to the end of the ring.

If you set `set-mark-command-repeat-pop` to non-`nil`, then immediately after you type `C-u C-SPC`, you can type `C-SPC` instead of `C-u C-SPC` to cycle through the mark ring. By default, `set-mark-command-repeat-pop` is `nil`.

Each buffer has its own mark ring. All editing commands use the current buffer's mark ring. In particular, `C-u C-SPC` always stays in the same buffer.

The variable `mark-ring-max` specifies the maximum number of entries to keep in the mark ring. This defaults to 16 entries. If that many entries exist and another one is pushed, the earliest one in the list is discarded. Repeating `C-u C-SPC` cycles through the positions currently in the ring.

If you want to move back to the same place over and over, the mark ring may not be convenient enough. If so, you can record the position in a register for later retrieval (see Section 10.1 [Saving Positions in Registers], page 106).

## 8.5 The Global Mark Ring

In addition to the ordinary mark ring that belongs to each buffer, Emacs has a single *global mark ring*. Each time you set a mark, this is recorded in the global mark ring in addition to the current buffer's own mark ring, if you have switched buffers since the previous mark setting. Hence, the global mark ring records a sequence of buffers that you have been in, and, for each buffer, a place where you set the mark. The length of the global mark ring is controlled by `global-mark-ring-max`, and is 16 by default.

The command `C-x C-SPC` (`pop-global-mark`) jumps to the buffer and position of the latest entry in the global ring. It also rotates the ring, so that successive uses of `C-x C-SPC` take you to earlier buffers and mark positions.

## 8.6 Shift Selection

If you hold down the shift key while typing a cursor motion command, this sets the mark before moving point, so that the region extends from the original position of point to its new position. This feature is referred to as *shift-selection*. It is similar to the way text is selected in other editors.

The mark set via shift-selection behaves a little differently from what we have described above. Firstly, in addition to the usual ways of deactivating the mark (such as changing the buffer text or typing `C-g`), the mark is deactivated by any *unshifted* cursor motion command. Secondly, any subsequent *shifted* cursor motion command avoids setting the mark anew. Therefore, a series of shifted cursor motion commands will continuously adjust the region.

Shift-selection only works if the shifted cursor motion key is not already bound to a separate command (see Chapter 33 [Customization], page 686). For example, if you bind `S-C-f` to another command, typing `S-C-f` runs that command instead of performing a shift-selected version of `C-f` (`forward-char`).

A mark set via mouse commands behaves the same as a mark set via shift-selection (see Section 8.1 [Setting Mark], page 89). For example, if you specify a region by dragging the

mouse, you can continue to extend the region using shifted cursor motion commands. In either case, any unshifted cursor motion command deactivates the mark.

To turn off shift-selection, set `shift-select-mode` to `nil`. Doing so does not disable setting the mark via mouse commands.

## 8.7 Disabling Transient Mark Mode

The default behavior of the mark and region, in which setting the mark activates it and highlights the region, is called Transient Mark mode. This is a minor mode that is enabled by default. It can be toggled with `M-x transient-mark-mode`, or with the '`Active Region Highlighting`' menu item in the '`Options`' menu. Turning it off switches Emacs to an alternative mode of operation:

- Setting the mark, with commands like `C-SPC` or `C-x C-x`, does not highlight the region. Therefore, you can't tell by looking where the mark is located; you have to remember.

  The usual solution to this problem is to set the mark and then use it soon, before you forget where it is. You can also check where the mark is by using `C-x C-x`, which exchanges the positions of the point and the mark (see Section 8.1 [Setting Mark], page 89).
- Some commands, which ordinarily act on the region when the mark is active, no longer do so. For example, normally `M-%` (`query-replace`) performs replacements within the region, if the mark is active. When Transient Mark mode is off, it always operates from point to the end of the buffer. Commands that act this way are identified in their own documentation.

While Transient Mark mode is off, you can activate it temporarily using `C-SPC C-SPC` or `C-u C-x C-x`.

`C-SPC C-SPC`
: Set the mark at point (like plain `C-SPC`) and enable Transient Mark mode just once, until the mark is deactivated. (This is not really a separate command; you are using the `C-SPC` command twice.)

`C-u C-x C-x`
: Exchange point and mark, activate the mark and enable Transient Mark mode temporarily, until the mark is next deactivated. (This is the `C-x C-x` command, `exchange-point-and-mark`, with a prefix argument.)

These commands set or activate the mark, and enable Transient Mark mode only until the mark is deactivated. One reason you may want to use them is that some commands operate on the entire buffer instead of the region when Transient Mark mode is off. Enabling Transient Mark mode momentarily gives you a way to use these commands on the region.

When you specify a region with the mouse (see Section 8.1 [Setting Mark], page 89), or with shift-selection (see Section 8.6 [Shift Selection], page 93), this likewise activates Transient Mark mode temporarily and highlights the region.

# 9 Killing and Moving Text

In Emacs, *killing* means erasing text and copying it into the *kill ring*. *Yanking* means bringing text from the kill ring back into the buffer. (Some applications use the terms "cutting" and "pasting" for similar operations.) The kill ring is so-named because it can be visualized as a set of blocks of text arranged in a ring, which you can access in cyclic order. See Section 9.2.1 [Kill Ring], page 98.

Killing and yanking are the most common way to move or copy text within Emacs. It is very versatile, because there are commands for killing many different types of syntactic units.

## 9.1 Deletion and Killing

Most commands which erase text from the buffer save it in the kill ring. These are known as *kill* commands, and their names normally contain the word '`kill`' (e.g., `kill-line`). The kill ring stores several recent kills, not just the last one, so killing is a very safe operation: you don't have to worry much about losing text that you previously killed. The kill ring is shared by all buffers, so text that is killed in one buffer can be yanked into another buffer.

When you use `C-/` (`undo`) to undo a kill command (see Section 22.9 [Undo], page 469), that brings the killed text back into the buffer, but does not remove it from the kill ring.

On graphical displays, killing text also copies it to the system clipboard. See Section 9.3 [Cut and Paste], page 100.

Commands that erase text but do not save it in the kill ring are known as *delete* commands; their names usually contain the word '`delete`'. These include `C-d` (`delete-char`) and DEL (`delete-backward-char`), which delete only one character at a time, and those commands that delete only spaces or newlines. Commands that can erase significant amounts of nontrivial data generally do a kill operation instead.

You can also use the mouse to kill and yank. See Section 9.3 [Cut and Paste], page 100.

### 9.1.1 Deletion

Deletion means erasing text and not saving it in the kill ring. For the most part, the Emacs commands that delete text are those that erase just one character or only whitespace.

DEL
BACKSPACE
: Delete the previous character, or the text in the region if it is active (`delete-backward-char`).

DELETE
: Delete the next character, or the text in the region if it is active (`delete-forward-char`).

`C-d`
: Delete the next character (`delete-char`).

`M-\`
: Delete spaces and tabs around point (`delete-horizontal-space`).

`M-SPC`
: Delete spaces and tabs around point, leaving one space (`just-one-space`).

`C-x C-o`
: Delete blank lines around the current line (`delete-blank-lines`).

`M-^` Join two lines by deleting the intervening newline, along with any indentation following it (`delete-indentation`).

We have already described the basic deletion commands DEL (`delete-backward-char`), DELETE (`delete-forward-char`), and `C-d` (`delete-char`). See Section 4.3 [Erasing], page 61. With a numeric argument, they delete the specified number of characters. If the numeric argument is omitted or one, they delete all the text in the region if it is active (see Section 8.3 [Using Region], page 91).

The other delete commands are those that delete only whitespace characters: spaces, tabs and newlines. `M-\` (`delete-horizontal-space`) deletes all the spaces and tab characters before and after point. With a prefix argument, this only deletes spaces and tab characters before point. `M-SPC` (`just-one-space`) does likewise but leaves a single space before point, regardless of the number of spaces that existed previously (even if there were none before). With a numeric argument *n*, it leaves *n* spaces before point if *n* is positive; if *n* is negative, it deletes newlines in addition to spaces and tabs, leaving *-n* spaces before point.

`C-x C-o` (`delete-blank-lines`) deletes all blank lines after the current line. If the current line is blank, it deletes all blank lines preceding the current line as well (leaving one blank line, the current line). On a solitary blank line, it deletes that line.

`M-^` (`delete-indentation`) joins the current line and the previous line, by deleting a newline and all surrounding spaces, usually leaving a single space. See Section 22.17 [Indentation], page 483.

### 9.1.2 Killing by Lines

`C-k` Kill rest of line or one or more lines (`kill-line`).

`C-S-backspace`
Kill an entire line at once (`kill-whole-line`)

The simplest kill command is `C-k` (`kill-line`). If used at the end of a line, it kills the line-ending newline character, merging the next line into the current one (thus, a blank line is entirely removed). Otherwise, `C-k` kills all the text from point up to the end of the line; if point was originally at the beginning of the line, this leaves the line blank.

Spaces and tabs at the end of the line are ignored when deciding which case applies. As long as point is after the last visible character in the line, you can be sure that `C-k` will kill the newline. To kill an entire non-blank line, go to the beginning and type `C-k` twice.

In this context, "line" means a logical text line, not a screen line (see Section 4.8 [Continuation Lines], page 63).

When `C-k` is given a positive argument *n*, it kills *n* lines and the newlines that follow them (text on the current line before point is not killed). With a negative argument $-n$, it kills *n* lines preceding the current line, together with the text on the current line before point. `C-k` with an argument of zero kills the text before point on the current line.

If the variable `kill-whole-line` is non-`nil`, `C-k` at the very beginning of a line kills the entire line including the following newline. This variable is normally `nil`.

`C-S-backspace` (`kill-whole-line`) kills a whole line including its newline, regardless of the position of point within the line. Note that many text terminals will prevent you from typing the key sequence `C-S-backspace`.

### 9.1.3 Other Kill Commands

`C-w` Kill the region (`kill-region`).

`M-w` Copy the region into the kill ring (`kill-ring-save`).

`M-d` Kill the next word (`kill-word`). See ⟨undefined⟩ [Words], page ⟨undefined⟩.

`M-DEL` Kill one word backwards (`backward-kill-word`).

`C-x DEL` Kill back to beginning of sentence (`backward-kill-sentence`). See ⟨undefined⟩ [Sentences], page ⟨undefined⟩.

`M-k` Kill to the end of the sentence (`kill-sentence`).

`C-M-k` Kill the following balanced expression (`kill-sexp`). See Section 23.4.1 [Expressions], page 521.

`M-z` *char* Kill through the next occurrence of *char* (`zap-to-char`).

One of the commonly-used kill commands is `C-w` (`kill-region`), which kills the text in the region (see Chapter 8 [Mark], page 89). Similarly, `M-w` (`kill-ring-save`) copies the text in the region into the kill ring without removing it from the buffer. If the mark is inactive when you type `C-w` or `M-w`, the command acts on the text between point and where you last set the mark (see Section 8.3 [Using Region], page 91).

Emacs also provides commands to kill specific syntactic units: words, with `M-DEL` and `M-d` (see ⟨undefined⟩ [Words], page ⟨undefined⟩); balanced expressions, with `C-M-k` (see Section 23.4.1 [Expressions], page 521); and sentences, with `C-x DEL` and `M-k` (see ⟨undefined⟩ [Sentences], page ⟨undefined⟩).

The command `M-z` (`zap-to-char`) combines killing with searching: it reads a character and kills from point up to (and including) the next occurrence of that character in the buffer. A numeric argument acts as a repeat count; a negative argument means to search backward and kill text before point.

### 9.1.4 Options for Killing

Some specialized buffers contain *read-only text*, which cannot be modified and therefore cannot be killed. The kill commands work specially in a read-only buffer: they move over text and copy it to the kill ring, without actually deleting it from the buffer. Normally, they also beep and display an error message when this happens. But if you set the variable `kill-read-only-ok` to a non-`nil` value, they just print a message in the echo area to explain why the text has not been erased.

If you change the variable `kill-do-not-save-duplicates` to a non-`nil` value, identical subsequent kills yield a single kill-ring entry, without duplication.

## 9.2 Yanking

*Yanking* means reinserting text previously killed. The usual way to move or copy text is to kill it and then yank it elsewhere.

`C-y` Yank the last kill into the buffer, at point (`yank`).

`M-y` Replace the text just yanked with an earlier batch of killed text (`yank-pop`). See Section 9.2.2 [Earlier Kills], page 98.

`C-M-w` Cause the following command, if it is a kill command, to append to the previous kill (`append-next-kill`). See Section 9.2.3 [Appending Kills], page 99.

The basic yanking command is `C-y` (`yank`). It inserts the most recent kill, leaving the cursor at the end of the inserted text. It also sets the mark at the beginning of the inserted text, without activating the mark; this lets you jump easily to that position, if you wish, with `C-u C-SPC` (see Section 8.4 [Mark Ring], page 92).

With a plain prefix argument (`C-u C-y`), the command instead leaves the cursor in front of the inserted text, and sets the mark at the end. Using any other prefix argument specifies an earlier kill; e.g., `C-u 4 C-y` reinserts the fourth most recent kill. See Section 9.2.2 [Earlier Kills], page 98.

On graphical displays, `C-y` first checks if another application has placed any text in the system clipboard more recently than the last Emacs kill. If so, it inserts the clipboard's text instead. Thus, Emacs effectively treats "cut" or "copy" clipboard operations performed in other applications like Emacs kills, except that they are not recorded in the kill ring. See Section 9.3 [Cut and Paste], page 100, for details.

### 9.2.1 The Kill Ring

The *kill ring* is a list of blocks of text that were previously killed. There is only one kill ring, shared by all buffers, so you can kill text in one buffer and yank it in another buffer. This is the usual way to move text from one buffer to another. (There are several other methods: for instance, you could store the text in a register; see Section 22.21 [Registers], page 507. See Section 9.4 [Accumulating Text], page 102, for some other ways to move text around.)

The maximum number of entries in the kill ring is controlled by the variable `kill-ring-max`. The default is 60. If you make a new kill when this limit has been reached, Emacs makes room by deleting the oldest entry in the kill ring.

The actual contents of the kill ring are stored in a variable named `kill-ring`; you can view the entire contents of the kill ring with `C-h v kill-ring`.

### 9.2.2 Yanking Earlier Kills

As explained in Section 22.8.3 [Yanking], page 465, you can use a numeric argument to `C-y` to yank text that is no longer the most recent kill. This is useful if you remember which kill ring entry you want. If you don't, you can use the `M-y` (`yank-pop`) command to cycle through the possibilities.

If the previous command was a yank command, `M-y` takes the text that was yanked and replaces it with the text from an earlier kill. So, to recover the text of the next-to-the-last kill, first use `C-y` to yank the last kill, and then use `M-y` to replace it with the previous kill. `M-y` is allowed only after a `C-y` or another `M-y`.

You can understand `M-y` in terms of a "last yank" pointer which points at an entry in the kill ring. Each time you kill, the "last yank" pointer moves to the newly made entry at the front of the ring. `C-y` yanks the entry which the "last yank" pointer points to. `M-y` moves the "last yank" pointer to a different entry, and the text in the buffer changes to match. Enough `M-y` commands can move the pointer to any entry in the ring, so you can get any entry into the buffer. Eventually the pointer reaches the end of the ring; the next `M-y` loops back around to the first entry again.

M-y moves the "last yank" pointer around the ring, but it does not change the order of the entries in the ring, which always runs from the most recent kill at the front to the oldest one still remembered.

M-y can take a numeric argument, which tells it how many entries to advance the "last yank" pointer by. A negative argument moves the pointer toward the front of the ring; from the front of the ring, it moves "around" to the last entry and continues forward from there.

Once the text you are looking for is brought into the buffer, you can stop doing M-y commands and it will stay there. It's just a copy of the kill ring entry, so editing it in the buffer does not change what's in the ring. As long as no new killing is done, the "last yank" pointer remains at the same place in the kill ring, so repeating C-y will yank another copy of the same previous kill.

When you call C-y with a numeric argument, that also sets the "last yank" pointer to the entry that it yanks.

### 9.2.3 Appending Kills

Normally, each kill command pushes a new entry onto the kill ring. However, two or more kill commands in a row combine their text into a single entry, so that a single C-y yanks all the text as a unit, just as it was before it was killed.

Thus, if you want to yank text as a unit, you need not kill all of it with one command; you can keep killing line after line, or word after word, until you have killed it all, and you can still get it all back at once.

Commands that kill forward from point add onto the end of the previous killed text. Commands that kill backward from point add text onto the beginning. This way, any sequence of mixed forward and backward kill commands puts all the killed text into one entry without rearrangement. Numeric arguments do not break the sequence of appending kills. For example, suppose the buffer contains this text:

```
This is a line ⋆of sample text.
```

with point shown by ⋆. If you type M-d M-DEL M-d M-DEL, killing alternately forward and backward, you end up with '`a line of sample`' as one entry in the kill ring, and '`This is text.`' in the buffer. (Note the double space between '`is`' and '`text`', which you can clean up with M-SPC or M-q.)

Another way to kill the same text is to move back two words with M-b M-b, then kill all four words forward with C-u M-d. This produces exactly the same results in the buffer and in the kill ring. M-f M-f C-u M-DEL kills the same text, all going backward; once again, the result is the same. The text in the kill ring entry always has the same order that it had in the buffer before you killed it.

If a kill command is separated from the last kill command by other commands (not just numeric arguments), it starts a new entry on the kill ring. But you can force it to append by first typing the command C-M-w (`append-next-kill`) right before it. The C-M-w tells the following command, if it is a kill command, to append the text it kills to the last killed text, instead of starting a new entry. With C-M-w, you can kill several separated pieces of text and accumulate them to be yanked back in one place.

A kill command following M-w (`kill-ring-save`) does not append to the text that M-w copied into the kill ring.

## 9.3 "Cut and Paste" Operations on Graphical Displays

In most graphical desktop environments, you can transfer data (usually text) between different applications using a system facility called the *clipboard*. On X, two other similar facilities are available: the primary selection and the secondary selection. When Emacs is run on a graphical display, its kill and yank commands integrate with these facilities, so that you can easily transfer text between Emacs and other graphical applications.

By default, Emacs uses UTF-8 as the coding system for inter-program text transfers. If you find that the pasted text is not what you expected, you can specify another coding system by typing `C-x RET x` or `C-x RET X`. You can also request a different data type by customizing `x-select-request-type`. See Section 19.11 [Communication Coding], page 387.

### 9.3.1 Using the Clipboard

The *clipboard* is the facility that most graphical applications use for "cutting and pasting". When the clipboard exists, the kill and yank commands in Emacs make use of it.

When you kill some text with a command such as `C-w` (`kill-region`), or copy it to the kill ring with a command such as `M-w` (`kill-ring-save`), that text is also put in the clipboard.

When an Emacs kill command puts text in the clipboard, the existing clipboard contents are normally lost. Optionally, you can change `save-interprogram-paste-before-kill` to `t`. Then Emacs will first save the clipboard to its kill ring, preventing you from losing the old clipboard data—at the risk of high memory consumption if that data turns out to be large.

Yank commands, such as `C-y` (`yank`), also use the clipboard. If another application "owns" the clipboard—i.e., if you cut or copied text there more recently than your last kill command in Emacs—then Emacs yanks from the clipboard instead of the kill ring.

Normally, rotating the kill ring with `M-y` (`yank-pop`) does not alter the clipboard. However, if you change `yank-pop-change-selection` to `t`, then `M-y` saves the new yank to the clipboard.

To prevent kill and yank commands from accessing the clipboard, change the variable `x-select-enable-clipboard` to `nil`.

Many X desktop environments support a feature called the *clipboard manager*. If you exit Emacs while it is the current "owner" of the clipboard data, and there is a clipboard manager running, Emacs transfers the clipboard data to the clipboard manager so that it is not lost. In some circumstances, this may cause a delay when exiting Emacs; if you wish to prevent Emacs from transferring data to the clipboard manager, change the variable `x-select-enable-clipboard-manager` to `nil`.

Prior to Emacs 24, the kill and yank commands used the primary selection (see Section 9.3.2 [Primary Selection], page 101), not the clipboard. If you prefer this behavior, change `x-select-enable-clipboard` to `nil`, `x-select-enable-primary` to `t`, and `mouse-drag-copy-region` to `t`. In this case, you can use the following commands to act explicitly on the clipboard: `clipboard-kill-region` kills the region and saves it to the clipboard; `clipboard-kill-ring-save` copies the region to the kill ring and saves it to the clipboard; and `clipboard-yank` yanks the contents of the clipboard at point.

### 9.3.2 Cut and Paste with Other Window Applications

Under the X Window System, there exists a *primary selection* containing the last stretch of text selected in an X application (usually by dragging the mouse). Typically, this text can be inserted into other X applications by `mouse-2` clicks. The primary selection is separate from the clipboard. Its contents are more "fragile"; they are overwritten each time you select text with the mouse, whereas the clipboard is only overwritten by explicit "cut" or "copy" commands.

Under X, whenever the region is active (see Chapter 8 [Mark], page 89), the text in the region is saved in the primary selection. This applies regardless of whether the region was made by dragging or clicking the mouse (see ⟨undefined⟩ [Mouse Commands], page ⟨undefined⟩), or by keyboard commands (e.g., by typing `C-SPC` and moving point; see Section 8.1 [Setting Mark], page 89).

If you change the variable `select-active-regions` to `only`, Emacs saves only temporarily active regions to the primary selection, i.e., those made with the mouse or with shift selection (see Section 8.6 [Shift Selection], page 93). If you change `select-active-regions` to `nil`, Emacs avoids saving active regions to the primary selection entirely.

To insert the primary selection into an Emacs buffer, click `mouse-2` (`mouse-yank-primary`) where you want to insert it. See ⟨undefined⟩ [Mouse Commands], page ⟨undefined⟩.

MS-Windows provides no primary selection, but Emacs emulates it within a single Emacs session by storing the selected text internally. Therefore, all the features and commands related to the primary selection work on Windows as they do on X, for cutting and pasting within the same session, but not across Emacs sessions or with other applications.

### 9.3.3 Secondary Selection

In addition to the primary selection, the X Window System provides a second similar facility known as the *secondary selection*. Nowadays, few X applications make use of the secondary selection, but you can access it using the following Emacs commands:

`M-Drag-Mouse-1`
: Set the secondary selection, with one end at the place where you press down the button, and the other end at the place where you release it (`mouse-set-secondary`). The selected text is highlighted, using the `secondary-selection` face, as you drag. The window scrolls automatically if you drag the mouse off the top or bottom of the window, just like `mouse-set-region` (see ⟨undefined⟩ [Mouse Commands], page ⟨undefined⟩).

  This command does not alter the kill ring.

`M-Mouse-1`
: Set one endpoint for the *secondary selection* (`mouse-start-secondary`).

`M-Mouse-3`
: Set the secondary selection, with one end at the position clicked and the other at the position specified with `M-Mouse-1` (`mouse-secondary-save-then-kill`). This also puts the selected text in the kill ring. A second `M-Mouse-3` at the same place kills the secondary selection just made.

`M-Mouse-2`
: Insert the secondary selection where you click, placing point at the end of the yanked text (`mouse-yank-secondary`).

Double or triple clicking of `M-Mouse-1` operates on words and lines, much like `Mouse-1`.

If `mouse-yank-at-point` is non-`nil`, `M-Mouse-2` yanks at point. Then it does not matter precisely where you click, or even which of the frame's windows you click on. See ⟨undefined⟩ [Mouse Commands], page ⟨undefined⟩.

## 9.4 Accumulating Text

Usually we copy or move text by killing it and yanking it, but there are other convenient methods for copying one block of text in many places, or for copying many scattered blocks of text into one place. Here we describe the commands to accumulate scattered pieces of text into a buffer or into a file.

`M-x append-to-buffer`
: Append region to the contents of a specified buffer.

`M-x prepend-to-buffer`
: Prepend region to the contents of a specified buffer.

`M-x copy-to-buffer`
: Copy region into a specified buffer, deleting that buffer's old contents.

`M-x insert-buffer`
: Insert the contents of a specified buffer into current buffer at point.

`M-x append-to-file`
: Append region to the contents of a specified file, at the end.

To accumulate text into a buffer, use `M-x append-to-buffer`. This reads a buffer name, then inserts a copy of the region into the buffer specified. If you specify a nonexistent buffer, `append-to-buffer` creates the buffer. The text is inserted wherever point is in that buffer. If you have been using the buffer for editing, the copied text goes into the middle of the text of the buffer, starting from wherever point happens to be at that moment.

Point in that buffer is left at the end of the copied text, so successive uses of `append-to-buffer` accumulate the text in the specified buffer in the same order as they were copied. Strictly speaking, `append-to-buffer` does not always append to the text already in the buffer—it appends only if point in that buffer is at the end. However, if `append-to-buffer` is the only command you use to alter a buffer, then point is always at the end.

`M-x prepend-to-buffer` is just like `append-to-buffer` except that point in the other buffer is left before the copied text, so successive prependings add text in reverse order. `M-x copy-to-buffer` is similar, except that any existing text in the other buffer is deleted, so the buffer is left containing just the text newly copied into it.

The command `M-x insert-buffer` can be used to retrieve the accumulated text from another buffer. This prompts for the name of a buffer, and inserts a copy of all the text in that buffer into the current buffer at point, leaving point at the beginning of the inserted text. It also adds the position of the end of the inserted text to the mark ring, without activating the mark. See Chapter 16 [Buffers], page 272, for background information on buffers.

Instead of accumulating text in a buffer, you can append text directly into a file with `M-x append-to-file`. This prompts for a filename, and adds the text of the region to the end of the specified file. The file is changed immediately on disk.

You should use `append-to-file` only with files that are *not* being visited in Emacs. Using it on a file that you are editing in Emacs would change the file behind Emacs's back, which can lead to losing some of your editing.

Another way to move text around is to store it in a register. See Section 22.21 [Registers], page 507.

## 9.5 Rectangles

*Rectangle* commands operate on rectangular areas of the text: all the characters between a certain pair of columns, in a certain range of lines. Emacs has commands to kill rectangles, yank killed rectangles, clear them out, fill them with blanks or text, or delete them. Rectangle commands are useful with text in multicolumn formats, and for changing text into or out of such formats.

To specify a rectangle for a command to work on, set the mark at one corner and point at the opposite corner. The rectangle thus specified is called the *region-rectangle*. If point and the mark are in the same column, the region-rectangle is empty. If they are in the same line, the region-rectangle is one line high.

The region-rectangle is controlled in much the same way as the region is controlled. But remember that a given combination of point and mark values can be interpreted either as a region or as a rectangle, depending on the command that uses them.

`C-x r k` Kill the text of the region-rectangle, saving its contents as the "last killed rectangle" (`kill-rectangle`).

`C-x r M-w` Save the text of the region-rectangle as the "last killed rectangle" (`copy-rectangle-as-kill`).

`C-x r d` Delete the text of the region-rectangle (`delete-rectangle`).

`C-x r y` Yank the last killed rectangle with its upper left corner at point (`yank-rectangle`).

`C-x r o` Insert blank space to fill the space of the region-rectangle (`open-rectangle`). This pushes the previous contents of the region-rectangle to the right.

`C-x r N` Insert line numbers along the left edge of the region-rectangle (`rectangle-number-lines`). This pushes the previous contents of the region-rectangle to the right.

`C-x r c` Clear the region-rectangle by replacing all of its contents with spaces (`clear-rectangle`).

`M-x delete-whitespace-rectangle`
Delete whitespace in each of the lines on the specified rectangle, starting from the left edge column of the rectangle.

`C-x r t` *string* `RET`
Replace rectangle contents with *string* on each line (`string-rectangle`).

`M-x string-insert-rectangle RET` *string* `RET`
: Insert *string* on each line of the rectangle.

The rectangle operations fall into two classes: commands to erase or insert rectangles, and commands to make blank rectangles.

There are two ways to erase the text in a rectangle: `C-x r d` (`delete-rectangle`) to delete the text outright, or `C-x r k` (`kill-rectangle`) to remove the text and save it as the *last killed rectangle*. In both cases, erasing the region-rectangle is like erasing the specified text on each line of the rectangle; if there is any following text on the line, it moves backwards to fill the gap.

"Killing" a rectangle is not killing in the usual sense; the rectangle is not stored in the kill ring, but in a special place that only records the most recent rectangle killed. This is because yanking a rectangle is so different from yanking linear text that different yank commands have to be used. Yank-popping is not defined for rectangles.

`C-x r M-w` (`copy-rectangle-as-kill`) is the equivalent of `M-w` for rectangles: it records the rectangle as the "last killed rectangle", without deleting the text from the buffer.

To yank the last killed rectangle, type `C-x r y` (`yank-rectangle`). The rectangle's first line is inserted at point, the rectangle's second line is inserted at the same horizontal position one line vertically below, and so on. The number of lines affected is determined by the height of the saved rectangle.

For example, you can convert two single-column lists into a double-column list by killing one of the single-column lists as a rectangle, and then yanking it beside the other list.

You can also copy rectangles into and out of registers with `C-x r r r` and `C-x r i r`. See Section 10.3 [Rectangle Registers], page 107.

There are two commands you can use for making blank rectangles: `C-x r c` (`clear-rectangle`) blanks out existing text in the region-rectangle, and `C-x r o` (`open-rectangle`) inserts a blank rectangle.

`M-x delete-whitespace-rectangle` deletes horizontal whitespace starting from a particular column. This applies to each of the lines in the rectangle, and the column is specified by the left edge of the rectangle. The right edge of the rectangle does not make any difference to this command.

The command `C-x r N` (`rectangle-number-lines`) inserts line numbers along the left edge of the region-rectangle. Normally, the numbering begins from 1 (for the first line of the rectangle). With a prefix argument, the command prompts for a number to begin from, and for a format string with which to print the numbers (see Section "Formatting Strings" in *The Emacs Lisp Reference Manual*).

The command `C-x r t` (`string-rectangle`) replaces the contents of a region-rectangle with a string on each line. The string's width need not be the same as the width of the rectangle. If the string's width is less, the text after the rectangle shifts left; if the string is wider than the rectangle, the text after the rectangle shifts right.

The command `M-x string-insert-rectangle` is similar to `string-rectangle`, but inserts the string on each line, shifting the original text to the right.

## 9.6 CUA Bindings

The command `M-x cua-mode` sets up key bindings that are compatible with the Common User Access (CUA) system used in many other applications.

When CUA mode is enabled, the keys `C-x`, `C-c`, `C-v`, and `C-z` invoke commands that cut (kill), copy, paste (yank), and undo respectively. The `C-x` and `C-c` keys perform cut and copy only if the region is active. Otherwise, they still act as prefix keys, so that standard Emacs commands like `C-x C-c` still work. Note that this means the variable `mark-even-if-inactive` has no effect for `C-x` and `C-c` (see Section 8.3 [Using Region], page 91).

To enter an Emacs command like `C-x C-f` while the mark is active, use one of the following methods: either hold `Shift` together with the prefix key, e.g., `S-C-x C-f`, or quickly type the prefix key twice, e.g., `C-x C-x C-f`.

To disable the overriding of standard Emacs binding by CUA mode, while retaining the other features of CUA mode described below, set the variable `cua-enable-cua-keys` to `nil`.

In CUA mode, typed text replaces the active region as in Delete-Selection mode (see ⟨undefined⟩ [Mouse Commands], page ⟨undefined⟩).

CUA mode provides enhanced rectangle support with visible rectangle highlighting. Use `C-RET` to start a rectangle, extend it using the movement commands, and cut or copy it using `C-x` or `C-c`. `RET` moves the cursor to the next (clockwise) corner of the rectangle, so you can easily expand it in any direction. Normal text you type is inserted to the left or right of each line in the rectangle (on the same side as the cursor).

With CUA you can easily copy text and rectangles into and out of registers by providing a one-digit numeric prefix to the kill, copy, and yank commands, e.g., `C-1 C-c` copies the region into register `1`, and `C-2 C-v` yanks the contents of register `2`.

CUA mode also has a global mark feature which allows easy moving and copying of text between buffers. Use `C-S-SPC` to toggle the global mark on and off. When the global mark is on, all text that you kill or copy is automatically inserted at the global mark, and text you type is inserted at the global mark rather than at the current position.

For example, to copy words from various buffers into a word list in a given buffer, set the global mark in the target buffer, then navigate to each of the words you want in the list, mark it (e.g., with `S-M-f`), copy it to the list with `C-c` or `M-w`, and insert a newline after the word in the target list by pressing RET.

# 10 Registers

Emacs *registers* are compartments where you can save text, rectangles, positions, and other things for later use. Once you save text or a rectangle in a register, you can copy it into the buffer once, or many times; once you save a position in a register, you can jump back to that position once, or many times.

Each register has a name that consists of a single character, which we will denote by *r*; *r* can be a letter (such as 'a') or a number (such as '1'); case matters, so register 'a' is not the same as register 'A'.

A register can store a position, a piece of text, a rectangle, a number, a window configuration, or a file name, but only one thing at any given time. Whatever you store in a register remains there until you store something else in that register. To see what register *r* contains, use `M-x view-register`:

`M-x view-register RET` *r*
: Display a description of what register *r* contains.

*Bookmarks* record files and positions in them, so you can return to those positions when you look at the file again. Bookmarks are similar in spirit to registers, so they are also documented in this chapter.

## 10.1 Saving Positions in Registers

`C-x r SPC` *r*
: Record the position of point and the current buffer in register *r* (`point-to-register`).

`C-x r j` *r* Jump to the position and buffer saved in register *r* (`jump-to-register`).

Typing `C-x r SPC` (`point-to-register`), followed by a character *r*, saves both the position of point and the current buffer in register *r*. The register retains this information until you store something else in it.

The command `C-x r j` *r* switches to the buffer recorded in register *r*, and moves point to the recorded position. The contents of the register are not changed, so you can jump to the saved position any number of times.

If you use `C-x r j` to go to a saved position, but the buffer it was saved from has been killed, `C-x r j` tries to create the buffer again by visiting the same file. Of course, this works only for buffers that were visiting files.

## 10.2 Saving Text in Registers

When you want to insert a copy of the same piece of text several times, it may be inconvenient to yank it from the kill ring, since each subsequent kill moves that entry further down the ring. An alternative is to store the text in a register and later retrieve it.

`C-x r s` *r* Copy region into register *r* (`copy-to-register`).

`C-x r i` *r* Insert text from register *r* (`insert-register`).

`M-x append-to-register RET r`
: Append region to text in register *r*.

  When register *r* contains text, you can use `C-x r +` (`increment-register`) to append to that register. Note that command `C-x r +` behaves differently if *r* contains a number. See Section 10.5 [Number Registers], page 108.

`M-x prepend-to-register RET r`
: Prepend region to text in register *r*.

`C-x r s r` stores a copy of the text of the region into the register named *r*. If the mark is inactive, Emacs first reactivates the mark where it was last set. The mark is deactivated at the end of this command. See Chapter 8 [Mark], page 89. `C-u C-x r s r`, the same command with a prefix argument, copies the text into register *r* and deletes the text from the buffer as well; you can think of this as "moving" the region text into the register.

`M-x append-to-register RET r` appends the copy of the text in the region to the text already stored in the register named *r*. If invoked with a prefix argument, it deletes the region after appending it to the register. The command `prepend-to-register` is similar, except that it *prepends* the region text to the text in the register instead of *appending* it.

When you are collecting text using `append-to-register` and `prepend-to-register`, you may want to separate individual collected pieces using a separator. In that case, configure a `register-separator` and store the separator text in to that register. For example, to get double newlines as text separator during the collection process, you can use the following setting.

```
(setq register-separator ?+)
(set-register register-separator "\n\n")
```

`C-x r i r` inserts in the buffer the text from register *r*. Normally it leaves point before the text and sets the mark after, without activating it. With a numeric argument, it instead puts point after the text and the mark before.

## 10.3 Saving Rectangles in Registers

A register can contain a rectangle instead of linear text. See Section 9.5 [Rectangles], page 103, for basic information on how to specify a rectangle in the buffer.

`C-x r r r` Copy the region-rectangle into register *r* (`copy-rectangle-to-register`). With numeric argument, delete it as well.

`C-x r i r` Insert the rectangle stored in register *r* (if it contains a rectangle) (`insert-register`).

The `C-x r i r` (`insert-register`) command, previously documented in Section 10.2 [Text Registers], page 106, inserts a rectangle rather than a text string, if the register contains a rectangle.

## 10.4 Saving Window Configurations in Registers

You can save the window configuration of the selected frame in a register, or even the configuration of all windows in all frames, and restore the configuration later. See Chapter 17 [Windows], page 289, for information about window configurations.

`C-x r w r` Save the state of the selected frame's windows in register *r* (`window-configuration-to-register`).

`C-x r f r` Save the state of all frames, including all their windows, in register *r* (`frame-configuration-to-register`).

Use `C-x r j r` to restore a window or frame configuration. This is the same command used to restore a cursor position. When you restore a frame configuration, any existing frames not included in the configuration become invisible. If you wish to delete these frames instead, use `C-u C-x r j r`.

## 10.5 Keeping Numbers in Registers

There are commands to store a number in a register, to insert the number in the buffer in decimal, and to increment it. These commands can be useful in keyboard macros (see Chapter 14 [Keyboard Macros], page 222).

`C-u` *number* `C-x r n r`
Store *number* into register *r* (`number-to-register`).

`C-u` *number* `C-x r + r`
If *r* contains a number, increment the number in that register by *number*. Note that command `C-x r +` (`increment-register`) behaves differently if *r* contains text. See Section 10.2 [Text Registers], page 106.

`C-x r i r` Insert the number from register *r* into the buffer.

`C-x r i` is the same command used to insert any other sort of register contents into the buffer. `C-x r +` with no numeric argument increments the register value by 1; `C-x r n` with no numeric argument stores zero in the register.

## 10.6 Keeping File Names in Registers

If you visit certain file names frequently, you can visit them more conveniently if you put their names in registers. Here's the Lisp code used to put a file name in a register:

```
(set-register ?r '(file . name))
```

For example,

```
(set-register ?z '(file . "/gd/gnu/emacs/19.0/src/ChangeLog"))
```

puts the file name shown in register 'z'.

To visit the file whose name is in register *r*, type `C-x r j r`. (This is the same command used to jump to a position or restore a frame configuration.)

## 10.7 Bookmarks

*Bookmarks* are somewhat like registers in that they record positions you can jump to. Unlike registers, they have long names, and they persist automatically from one Emacs session to the next. The prototypical use of bookmarks is to record "where you were reading" in various files.

`C-x r m RET`
: Set the bookmark for the visited file, at point.

`C-x r m` *bookmark* `RET`
: Set the bookmark named *bookmark* at point (`bookmark-set`).

`C-x r b` *bookmark* `RET`
: Jump to the bookmark named *bookmark* (`bookmark-jump`).

`C-x r l`
: List all bookmarks (`list-bookmarks`).

`M-x bookmark-save`
: Save all the current bookmark values in the default bookmark file.

The prototypical use for bookmarks is to record one current position in each of several files. So the command `C-x r m`, which sets a bookmark, uses the visited file name as the default for the bookmark name. If you name each bookmark after the file it points to, then you can conveniently revisit any of those files with `C-x r b`, and move to the position of the bookmark at the same time.

To display a list of all your bookmarks in a separate buffer, type `C-x r l` (`list-bookmarks`). If you switch to that buffer, you can use it to edit your bookmark definitions or annotate the bookmarks. Type `C-h m` in the bookmark buffer for more information about its special editing commands.

When you kill Emacs, Emacs saves your bookmarks, if you have changed any bookmark values. You can also save the bookmarks at any time with the `M-x bookmark-save` command. Bookmarks are saved to the file '`~/.emacs.d/bookmarks`' (for compatibility with older versions of Emacs, if you have a file named '`~/.emacs.bmk`', that is used instead). The bookmark commands load your default bookmark file automatically. This saving and loading is how bookmarks persist from one Emacs session to the next.

If you set the variable `bookmark-save-flag` to 1, each command that sets a bookmark will also save your bookmarks; this way, you don't lose any bookmark values even if Emacs crashes. The value, if a number, says how many bookmark modifications should go by between saving. If you set this variable to `nil`, Emacs only saves bookmarks if you explicitly use `M-x bookmark-save`.

Bookmark position values are saved with surrounding context, so that `bookmark-jump` can find the proper position even if the file is modified slightly. The variable `bookmark-`

`search-size` says how many characters of context to record on each side of the bookmark's position.

Here are some additional commands for working with bookmarks:

`M-x bookmark-load RET` *`filename`* `RET`
: Load a file named *filename* that contains a list of bookmark values. You can use this command, as well as `bookmark-write`, to work with other files of bookmark values in addition to your default bookmark file.

`M-x bookmark-write RET` *`filename`* `RET`
: Save all the current bookmark values in the file *filename*.

`M-x bookmark-delete RET` *`bookmark`* `RET`
: Delete the bookmark named *bookmark*.

`M-x bookmark-insert-location RET` *`bookmark`* `RET`
: Insert in the buffer the name of the file that bookmark *bookmark* points to.

`M-x bookmark-insert RET` *`bookmark`* `RET`
: Insert in the buffer the *contents* of the file that bookmark *bookmark* points to.

# 11 Emacs Display

This chapter describes a number of features related to the display that Emacs presents to the user.

## 11.1 Refreshing the Screen

The function `redraw-frame` clears and redisplays the entire contents of a given frame (see Chapter 18 [Frames], page 341). This is useful if the screen is corrupted.

`redraw-frame` *frame* [Function]
: This function clears and redisplays frame *frame*.

Even more powerful is `redraw-display`:

`redraw-display` [Command]
: This function clears and redisplays all visible frames.

In Emacs, processing user input takes priority over redisplay. If you call these functions when input is available, they don't redisplay immediately, but the requested redisplay does happen eventually—after all the input has been processed.

On text terminals, suspending and resuming Emacs normally also refreshes the screen. Some terminal emulators record separate contents for display-oriented programs such as Emacs and for ordinary sequential display. If you are using such a terminal, you might want to inhibit the redisplay on resumption.

`no-redraw-on-reenter` [User Option]
: This variable controls whether Emacs redraws the entire screen after it has been suspended and resumed. Non-`nil` means there is no need to redraw, `nil` means redrawing is needed. The default is `nil`.

## 11.2 Forcing Redisplay

Emacs normally tries to redisplay the screen whenever it waits for input. With the following function, you can request an immediate attempt to redisplay, in the middle of Lisp code, without actually waiting for input.

`redisplay` **&optional** *force* [Function]
: This function tries immediately to redisplay. The optional argument *force*, if non-`nil`, forces the redisplay to be performed, instead of being preempted, even if input is pending and the variable `redisplay-dont-pause` is `nil` (see below). If `redisplay-dont-pause` is non-`nil` (the default), this function redisplays in any case, i.e., *force* does nothing.

  The function returns `t` if it actually tried to redisplay, and `nil` otherwise. A value of `t` does not mean that redisplay proceeded to completion; it could have been preempted by newly arriving input.

`redisplay-dont-pause` [Variable]
: If this variable is `nil`, arriving input events preempt redisplay; Emacs avoids starting a redisplay, and stops any redisplay that is in progress, until the input has been

processed. In particular, `(redisplay)` returns `nil` without actually redisplaying, if there is pending input.

The default value is `t`, which means that pending input does not preempt redisplay.

`redisplay-preemption-period` [Variable]

If `redisplay-dont-pause` is `nil`, this variable specifies how many seconds Emacs waits between checks for new input during redisplay; if input arrives during this interval, redisplay stops and the input is processed. The default value is 0.1; if the value is `nil`, Emacs does not check for input during redisplay.

This variable has no effect when `redisplay-dont-pause` is non-`nil` (the default).

Although `redisplay` tries immediately to redisplay, it does not change how Emacs decides which parts of its frame(s) to redisplay. By contrast, the following function adds certain windows to the pending redisplay work (as if their contents had completely changed), but does not immediately try to perform redisplay.

`force-window-update` **&optional** *object* [Function]

This function forces some or all windows to be updated the next time Emacs does a redisplay. If *object* is a window, that window is to be updated. If *object* is a buffer or buffer name, all windows displaying that buffer are to be updated. If *object* is `nil` (or omitted), all windows are to be updated.

This function does not do a redisplay immediately; Emacs does that as it waits for input, or when the function `redisplay` is called.

## 11.3 Truncation

When a line of text extends beyond the right edge of a window, Emacs can *continue* the line (make it "wrap" to the next screen line), or *truncate* the line (limit it to one screen line). The additional screen lines used to display a long text line are called *continuation* lines. Continuation is not the same as filling; continuation happens on the screen only, not in the buffer contents, and it breaks a line precisely at the right margin, not at a word boundary. See Section 22.11 [Filling], page 473.

On a graphical display, tiny arrow images in the window fringes indicate truncated and continued lines (see Section 11.13 [Fringes], page 156). On a text terminal, a '$' in the rightmost column of the window indicates truncation; a '\' on the rightmost column indicates a line that "wraps". (The display table can specify alternate characters to use for this; see Section 11.20.2 [Display Tables], page 190).

`truncate-lines` [User Option]

If this buffer-local variable is non-`nil`, lines that extend beyond the right edge of the window are truncated; otherwise, they are continued. As a special exception, the variable `truncate-partial-width-windows` takes precedence in *partial-width* windows (i.e., windows that do not occupy the entire frame width).

`truncate-partial-width-windows` [User Option]

This variable controls line truncation in *partial-width* windows. A partial-width window is one that does not occupy the entire frame width (see Section 17.5 [Splitting

Windows], page 297). If the value is `nil`, line truncation is determined by the variable `truncate-lines` (see above). If the value is an integer *n*, lines are truncated if the partial-width window has fewer than *n* columns, regardless of the value of `truncate-lines`; if the partial-width window has *n* or more columns, line truncation is determined by `truncate-lines`. For any other non-`nil` value, lines are truncated in every partial-width window, regardless of the value of `truncate-lines`.

When horizontal scrolling (see Section 17.22 [Horizontal Scrolling], page 331) is in use in a window, that forces truncation.

`wrap-prefix` [Variable]

If this buffer-local variable is non-`nil`, it defines a *wrap prefix* which Emacs displays at the start of every continuation line. (If lines are truncated, `wrap-prefix` is never used.) Its value may be a string or an image (see Section 11.15.4 [Other Display Specs], page 166), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 11.15.2 [Specified Space], page 164). The value is interpreted in the same way as a `display` text property. See Section 11.15 [Display Property], page 163.

A wrap prefix may also be specified for regions of text, using the `wrap-prefix` text or overlay property. This takes precedence over the `wrap-prefix` variable. See Section 22.19.4 [Special Properties], page 494.

`line-prefix` [Variable]

If this buffer-local variable is non-`nil`, it defines a *line prefix* which Emacs displays at the start of every non-continuation line. Its value may be a string or an image (see Section 11.15.4 [Other Display Specs], page 166), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 11.15.2 [Specified Space], page 164). The value is interpreted in the same way as a `display` text property. See Section 11.15 [Display Property], page 163.

A line prefix may also be specified for regions of text using the `line-prefix` text or overlay property. This takes precedence over the `line-prefix` variable. See Section 22.19.4 [Special Properties], page 494.

If your buffer contains *very* long lines, and you use continuation to display them, computing the continuation lines can make redisplay slow. The column computation and indentation functions also become slow. Then you might find it advisable to set `cache-long-line-scans` to `t`.

`cache-long-line-scans` [Variable]

If this variable is non-`nil`, various indentation and motion functions, and Emacs redisplay, cache the results of scanning the buffer, and consult the cache to avoid rescanning regions of the buffer unless they are modified.

Turning on the cache slows down processing of short lines somewhat.

This variable is automatically buffer-local in every buffer.

## 11.4 The Echo Area

The *echo area* is used for displaying error messages (see ⟨undefined⟩ [Errors], page ⟨undefined⟩), for messages made with the `message` primitive, and for echoing keystrokes. It is not the same as the minibuffer, despite the fact that the minibuffer appears (when active) in the same place on the screen as the echo area. See Section "The Minibuffer" in *The GNU Emacs Manual*.

Apart from the functions documented in this section, you can print Lisp objects to the echo area by specifying `t` as the output stream. See ⟨undefined⟩ [Output Streams], page ⟨undefined⟩.

### 11.4.1 Displaying Messages in the Echo Area

This section describes the standard functions for displaying messages in the echo area.

`message` *format-string* **&rest** *arguments* [Function]

This function displays a message in the echo area. *format-string* is a format string, and *arguments* are the objects for its format specifications, like in the `format` function (see ⟨undefined⟩ [Formatting Strings], page ⟨undefined⟩). The resulting formatted string is displayed in the echo area; if it contains `face` text properties, it is displayed with the specified faces (see Section 11.12 [Faces], page 137). The string is also added to the '`*Messages*`' buffer, but without text properties (see Section 11.4.3 [Logging Messages], page 117).

In batch mode, the message is printed to the standard error stream, followed by a newline.

If *format-string* is `nil` or the empty string, `message` clears the echo area; if the echo area has been expanded automatically, this brings it back to its normal size. If the minibuffer is active, this brings the minibuffer contents back onto the screen immediately.

```
(message "Minibuffer depth is %d."
         (minibuffer-depth))
 ⊣ Minibuffer depth is 0.
⇒ "Minibuffer depth is 0."

---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

To automatically display a message in the echo area or in a pop-buffer, depending on its size, use `display-message-or-buffer` (see below).

`with-temp-message` *message* **&rest** *body* [Macro]

This construct displays a message in the echo area temporarily, during the execution of *body*. It displays *message*, executes *body*, then returns the value of the last body form while restoring the previous echo area contents.

`message-or-box` *format-string* **&rest** *arguments* [Function]

This function displays a message like `message`, but may display it in a dialog box instead of the echo area. If this function is called in a command that was invoked

using the mouse—more precisely, if `last-nonmenu-event` (see Section 2.5 [Command Loop Info], page 20) is either `nil` or a list—then it uses a dialog box or pop-up menu to display the message. Otherwise, it uses the echo area. (This is the same criterion that `y-or-n-p` uses to make a similar decision; see ⟨undefined⟩ [Yes-or-No Queries], page ⟨undefined⟩.)

You can force use of the mouse or of the echo area by binding `last-nonmenu-event` to a suitable value around the call.

`message-box` *format-string* **&rest** *arguments* [Function]
This function displays a message like `message`, but uses a dialog box (or a pop-up menu) whenever that is possible. If it is impossible to use a dialog box or pop-up menu, because the terminal does not support them, then `message-box` uses the echo area, like `message`.

`display-message-or-buffer` *message* **&optional** *buffer-name* *not-this-window* *frame* [Function]
This function displays the message *message*, which may be either a string or a buffer. If it is shorter than the maximum height of the echo area, as defined by `max-mini-window-height`, it is displayed in the echo area, using `message`. Otherwise, `display-buffer` is used to show it in a pop-up buffer.

Returns either the string shown in the echo area, or when a pop-up buffer is used, the window used to display it.

If *message* is a string, then the optional argument *buffer-name* is the name of the buffer used to display it when a pop-up buffer is used, defaulting to '`*Message*`'. In the case where *message* is a string and displayed in the echo area, it is not specified whether the contents are inserted into the buffer anyway.

The optional arguments *not-this-window* and *frame* are as for `display-buffer`, and only used if a buffer is displayed.

`current-message` [Function]
This function returns the message currently being displayed in the echo area, or `nil` if there is none.

### 11.4.2 Reporting Operation Progress

When an operation can take a while to finish, you should inform the user about the progress it makes. This way the user can estimate remaining time and clearly see that Emacs is busy working, not hung. A convenient way to do this is to use a *progress reporter*.

Here is a working example that does nothing useful:

```
(let ((progress-reporter
       (make-progress-reporter "Collecting mana for Emacs..."
                               0  500)))
  (dotimes (k 500)
    (sit-for 0.01)
    (progress-reporter-update progress-reporter k))
  (progress-reporter-done progress-reporter))
```

`make-progress-reporter` *message* **&optional** *min-value max-value current-value min-change min-time* [Function]

This function creates and returns a progress reporter object, which you will use as an argument for the other functions listed below. The idea is to precompute as much data as possible to make progress reporting very fast.

When this progress reporter is subsequently used, it will display *message* in the echo area, followed by progress percentage. *message* is treated as a simple string. If you need it to depend on a filename, for instance, use `format` before calling this function.

The arguments *min-value* and *max-value* should be numbers standing for the starting and final states of the operation. For instance, an operation that "scans" a buffer should set these to the results of `point-min` and `point-max` correspondingly. *max-value* should be greater than *min-value*.

Alternatively, you can set *min-value* and *max-value* to `nil`. In that case, the progress reporter does not report process percentages; it instead displays a "spinner" that rotates a notch each time you update the progress reporter.

If *min-value* and *max-value* are numbers, you can give the argument *current-value* a numerical value specifying the initial progress; if omitted, this defaults to *min-value*.

The remaining arguments control the rate of echo area updates. The progress reporter will wait for at least *min-change* more percents of the operation to be completed before printing next message; the default is one percent. *min-time* specifies the minimum time in seconds to pass between successive prints; the default is 0.2 seconds. (On some operating systems, the progress reporter may handle fractions of seconds with varying precision).

This function calls `progress-reporter-update`, so the first message is printed immediately.

`progress-reporter-update` *reporter* **&optional** *value* [Function]

This function does the main work of reporting progress of your operation. It displays the message of *reporter*, followed by progress percentage determined by *value*. If percentage is zero, or close enough according to the *min-change* and *min-time* arguments, then it is omitted from the output.

*reporter* must be the result of a call to `make-progress-reporter`. *value* specifies the current state of your operation and must be between *min-value* and *max-value* (inclusive) as passed to `make-progress-reporter`. For instance, if you scan a buffer, then *value* should be the result of a call to `point`.

This function respects *min-change* and *min-time* as passed to `make-progress-reporter` and so does not output new messages on every invocation. It is thus very fast and normally you should not try to reduce the number of calls to it: resulting overhead will most likely negate your effort.

`progress-reporter-force-update` *reporter* **&optional** *value new-message* [Function]

This function is similar to `progress-reporter-update` except that it prints a message in the echo area unconditionally.

The first two arguments have the same meaning as for `progress-reporter-update`. Optional *new-message* allows you to change the message of the *reporter*. Since this

functions always updates the echo area, such a change will be immediately presented to the user.

`progress-reporter-done` *reporter* [Function]
This function should be called when the operation is finished. It prints the message of *reporter* followed by word "done" in the echo area.

You should always call this function and not hope for `progress-reporter-update` to print "100%". Firstly, it may never print it, there are many good reasons for this not to happen. Secondly, "done" is more explicit.

`dotimes-with-progress-reporter` (*var count* [*result*]) *message body*... [Macro]
This is a convenience macro that works the same way as `dotimes` does, but also reports loop progress using the functions described above. It allows you to save some typing.

You can rewrite the example in the beginning of this node using this macro this way:

```
(dotimes-with-progress-reporter
    (k 500)
    "Collecting some mana for Emacs..."
  (sit-for 0.01))
```

### 11.4.3 Logging Messages in '*Messages*'

Almost all the messages displayed in the echo area are also recorded in the '`*Messages*`' buffer so that the user can refer back to them. This includes all the messages that are output with `message`.

`message-log-max` [User Option]
This variable specifies how many lines to keep in the '`*Messages*`' buffer. The value `t` means there is no limit on how many lines to keep. The value `nil` disables message logging entirely. Here's how to display a message and prevent it from being logged:

```
(let (message-log-max)
  (message ...))
```

To make '`*Messages*`' more convenient for the user, the logging facility combines successive identical messages. It also combines successive related messages for the sake of two cases: question followed by answer, and a series of progress messages.

A "question followed by an answer" means two messages like the ones produced by `y-or-n-p`: the first is '*`question`*', and the second is '*`question...answer`*'. The first message conveys no additional information beyond what's in the second, so logging the second message discards the first from the log.

A "series of progress messages" means successive messages like those produced by `make-progress-reporter`. They have the form '*`base...how-far`*', where *base* is the same each time, while *how-far* varies. Logging each message in the series discards the previous one, provided they are consecutive.

The functions `make-progress-reporter` and `y-or-n-p` don't have to do anything special to activate the message log combination feature. It operates whenever two consecutive messages are logged that share a common prefix ending in '`...`'.

### 11.4.4 Echo Area Customization

These variables control details of how the echo area works.

`cursor-in-echo-area` [Variable]
: This variable controls where the cursor appears when a message is displayed in the echo area. If it is non-`nil`, then the cursor appears at the end of the message. Otherwise, the cursor appears at point—not in the echo area at all.

  The value is normally `nil`; Lisp programs bind it to `t` for brief periods of time.

`echo-area-clear-hook` [Variable]
: This normal hook is run whenever the echo area is cleared—either by `(message nil)` or for any other reason.

`echo-keystrokes` [User Option]
: This variable determines how much time should elapse before command characters echo. Its value must be an integer or floating point number, which specifies the number of seconds to wait before echoing. If the user types a prefix key (such as `C-x`) and then delays this many seconds before continuing, the prefix key is echoed in the echo area. (Once echoing begins in a key sequence, all subsequent characters in the same key sequence are echoed immediately.)

  If the value is zero, then command input is not echoed.

`message-truncate-lines` [Variable]
: Normally, displaying a long message resizes the echo area to display the entire message. But if the variable `message-truncate-lines` is non-`nil`, the echo area does not resize, and the message is truncated to fit it.

The variable `max-mini-window-height`, which specifies the maximum height for resizing minibuffer windows, also applies to the echo area (which is really a special use of the minibuffer window; see ⟨undefined⟩ [Minibuffer Misc], page ⟨undefined⟩).

## 11.5 Reporting Warnings

*Warnings* are a facility for a program to inform the user of a possible problem, but continue running.

### 11.5.1 Warning Basics

Every warning has a textual message, which explains the problem for the user, and a *severity level* which is a symbol. Here are the possible severity levels, in order of decreasing severity, and their meanings:

`:emergency`
: A problem that will seriously impair Emacs operation soon if you do not attend to it promptly.

`:error`
: A report of data or circumstances that are inherently wrong.

`:warning`
: A report of data or circumstances that are not inherently wrong, but raise suspicion of a possible problem.

`:debug` A report of information that may be useful if you are debugging.

When your program encounters invalid input data, it can either signal a Lisp error by calling `error` or `signal` or report a warning with severity `:error`. Signaling a Lisp error is the easiest thing to do, but it means the program cannot continue processing. If you want to take the trouble to implement a way to continue processing despite the bad data, then reporting a warning of severity `:error` is the right way to inform the user of the problem. For instance, the Emacs Lisp byte compiler can report an error that way and continue compiling other functions. (If the program signals a Lisp error and then handles it with `condition-case`, the user won't see the error message; it could show the message to the user by reporting it as a warning.)

Each warning has a *warning type* to classify it. The type is a list of symbols. The first symbol should be the custom group that you use for the program's user options. For example, byte compiler warnings use the warning type `(bytecomp)`. You can also subcategorize the warnings, if you wish, by using more symbols in the list.

`display-warning` *type message* **&optional** *level buffer-name* [Function]
: This function reports a warning, using *message* as the message and *type* as the warning type. *level* should be the severity level, with `:warning` being the default.

  *buffer-name*, if non-`nil`, specifies the name of the buffer for logging the warning. By default, it is '`*Warnings*`'.

`lwarn` *type level message* **&rest** *args* [Function]
: This function reports a warning using the value of `(format` *`message args...`*`)` as the message. In other respects it is equivalent to `display-warning`.

`warn` *message* **&rest** *args* [Function]
: This function reports a warning using the value of `(format` *`message args...`*`)` as the message, `(emacs)` as the type, and `:warning` as the severity level. It exists for compatibility only; we recommend not using it, because you should specify a specific warning type.

### 11.5.2 Warning Variables

Programs can customize how their warnings appear by binding the variables described in this section.

`warning-levels` [Variable]
: This list defines the meaning and severity order of the warning severity levels. Each element defines one severity level, and they are arranged in order of decreasing severity.

  Each element has the form `(`*`level string function`*`)`, where *level* is the severity level it defines. *string* specifies the textual description of this level. *string* should use '`%s`' to specify where to put the warning type information, or it can omit the '`%s`' so as not to include that information.

  The optional *function*, if non-`nil`, is a function to call with no arguments, to get the user's attention.

  Normally you should not change the value of this variable.

`warning-prefix-function` [Variable]

If non-`nil`, the value is a function to generate prefix text for warnings. Programs can bind the variable to a suitable function. `display-warning` calls this function with the warnings buffer current, and the function can insert text in it. That text becomes the beginning of the warning message.

The function is called with two arguments, the severity level and its entry in `warning-levels`. It should return a list to use as the entry (this value need not be an actual member of `warning-levels`). By constructing this value, the function can change the severity of the warning, or specify different handling for a given severity level.

If the variable's value is `nil` then there is no function to call.

`warning-series` [Variable]

Programs can bind this variable to `t` to say that the next warning should begin a series. When several warnings form a series, that means to leave point on the first warning of the series, rather than keep moving it for each warning so that it appears on the last one. The series ends when the local binding is unbound and `warning-series` becomes `nil` again.

The value can also be a symbol with a function definition. That is equivalent to `t`, except that the next warning will also call the function with no arguments with the warnings buffer current. The function can insert text which will serve as a header for the series of warnings.

Once a series has begun, the value is a marker which points to the buffer position in the warnings buffer of the start of the series.

The variable's normal value is `nil`, which means to handle each warning separately.

`warning-fill-prefix` [Variable]

When this variable is non-`nil`, it specifies a fill prefix to use for filling each warning's text.

`warning-type-format` [Variable]

This variable specifies the format for displaying the warning type in the warning message. The result of formatting the type this way gets included in the message under the control of the string in the entry in `warning-levels`. The default value is `" (%s)"`. If you bind it to `""` then the warning type won't appear at all.

### 11.5.3 Warning Options

These variables are used by users to control what happens when a Lisp program reports a warning.

`warning-minimum-level` [User Option]

This user option specifies the minimum severity level that should be shown immediately to the user. The default is `:warning`, which means to immediately display all warnings except `:debug` warnings.

`warning-minimum-log-level` [User Option]

This user option specifies the minimum severity level that should be logged in the warnings buffer. The default is `:warning`, which means to log all warnings except `:debug` warnings.

`warning-suppress-types` [User Option]
: This list specifies which warning types should not be displayed immediately for the user. Each element of the list should be a list of symbols. If its elements match the first elements in a warning type, then that warning is not displayed immediately.

`warning-suppress-log-types` [User Option]
: This list specifies which warning types should not be logged in the warnings buffer. Each element of the list should be a list of symbols. If it matches the first few elements in a warning type, then that warning is not logged.

### 11.5.4 Delayed Warnings

Sometimes, you may wish to avoid showing a warning while a command is running, and only show it only after the end of the command. You can use the variable `delayed-warnings-list` for this.

`delayed-warnings-list` [Variable]
: The value of this variable is a list of warnings to be displayed after the current command has finished. Each element must be a list

```
(type message [level [buffer-name]])
```

with the same form, and the same meanings, as the argument list of `display-warning` (see Section 11.5.1 [Warning Basics], page 118). Immediately after running `post-command-hook` (see Section 2.1 [Command Overview], page 11), the Emacs command loop displays all the warnings specified by this variable, then resets it to `nil`.

Programs which need to further customize the delayed warnings mechanism can change the variable `delayed-warnings-hook`:

`delayed-warnings-hook` [Variable]
: This is a normal hook which is run by the Emacs command loop, after `post-command-hook`, in order to to process and display delayed warnings.

Its default value is a list of two functions:

```
(collapse-delayed-warnings display-delayed-warnings)
```

The function `collapse-delayed-warnings` removes repeated entries from `delayed-warnings-list`. The function `display-delayed-warnings` calls `display-warning` on each of the entries in `delayed-warnings-list`, in turn, and then sets `delayed-warnings-list` to `nil`.

## 11.6 Invisible Text

You can make characters *invisible*, so that they do not appear on the screen, with the `invisible` property. This can be either a text property (see Section 22.19 [Text Properties], page 489) or an overlay property (see Section 11.9 [Overlays], page 128). Cursor motion also partly ignores these characters; if the command loop finds that point is inside a range of invisible text after a command, it relocates point to the other side of the text.

In the simplest case, any non-`nil` `invisible` property makes a character invisible. This is the default case—if you don't alter the default value of `buffer-invisibility-spec`, this is how the `invisible` property works. You should normally use `t` as the value of the `invisible` property if you don't plan to set `buffer-invisibility-spec` yourself.

More generally, you can use the variable `buffer-invisibility-spec` to control which values of the `invisible` property make text invisible. This permits you to classify the text into different subsets in advance, by giving them different `invisible` values, and subsequently make various subsets visible or invisible by changing the value of `buffer-invisibility-spec`.

Controlling visibility with `buffer-invisibility-spec` is especially useful in a program to display the list of entries in a database. It permits the implementation of convenient filtering commands to view just a part of the entries in the database. Setting this variable is very fast, much faster than scanning all the text in the buffer looking for properties to change.

`buffer-invisibility-spec` [Variable]

This variable specifies which kinds of `invisible` properties actually make a character invisible. Setting this variable makes it buffer-local.

`t` A character is invisible if its `invisible` property is non-`nil`. This is the default.

a list Each element of the list specifies a criterion for invisibility; if a character's `invisible` property fits any one of these criteria, the character is invisible. The list can have two kinds of elements:

*`atom`* A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member; comparison is done with `eq`.

`(`*`atom`*` . t)`
A character is invisible if its `invisible` property value is *atom* or if it is a list with *atom* as a member; comparison is done with `eq`. Moreover, a sequence of such characters displays as an ellipsis.

Two functions are specifically provided for adding elements to `buffer-invisibility-spec` and removing elements from it.

`add-to-invisibility-spec` *element* [Function]

This function adds the element *element* to `buffer-invisibility-spec`. If `buffer-invisibility-spec` was `t`, it changes to a list, `(t)`, so that text whose `invisible` property is `t` remains invisible.

`remove-from-invisibility-spec` *element* [Function]

This removes the element *element* from `buffer-invisibility-spec`. This does nothing if *element* is not in the list.

A convention for use of `buffer-invisibility-spec` is that a major mode should use the mode's own name as an element of `buffer-invisibility-spec` and as the value of the `invisible` property:

```
;; If you want to display an ellipsis:
(add-to-invisibility-spec '(my-symbol . t))
;; If you don't want ellipsis:
```

```
(add-to-invisibility-spec 'my-symbol)

(overlay-put (make-overlay beginning end)
             'invisible 'my-symbol)

;; When done with the invisibility:
(remove-from-invisibility-spec '(my-symbol . t))
;; Or respectively:
(remove-from-invisibility-spec 'my-symbol)
```

You can check for invisibility using the following function:

`invisible-p` *pos-or-prop* [Function]

If *pos-or-prop* is a marker or number, this function returns a non-`nil` value if the text at that position is invisible.

If *pos-or-prop* is any other kind of Lisp object, that is taken to mean a possible value of the `invisible` text or overlay property. In that case, this function returns a non-`nil` value if that value would cause text to become invisible, based on the current value of `buffer-invisibility-spec`.

Ordinarily, functions that operate on text or move point do not care whether the text is invisible. The user-level line motion commands ignore invisible newlines if `line-move-ignore-invisible` is non-`nil` (the default), but only because they are explicitly programmed to do so.

However, if a command ends with point inside or at the boundary of invisible text, the main editing loop relocates point to one of the two ends of the invisible text. Emacs chooses the direction of relocation so that it is the same as the overall movement direction of the command; if in doubt, it prefers a position where an inserted char would not inherit the `invisible` property. Additionally, if the text is not replaced by an ellipsis and the command only moved within the invisible text, then point is moved one extra character so as to try and reflect the command's movement by a visible movement of the cursor.

Thus, if the command moved point back to an invisible range (with the usual stickiness), Emacs moves point back to the beginning of that range. If the command moved point forward into an invisible range, Emacs moves point forward to the first visible character that follows the invisible text and then forward one more character.

Incremental search can make invisible overlays visible temporarily and/or permanently when a match includes invisible text. To enable this, the overlay should have a non-`nil` `isearch-open-invisible` property. The property value should be a function to be called with the overlay as an argument. This function should make the overlay visible permanently; it is used when the match overlaps the overlay on exit from the search.

During the search, such overlays are made temporarily visible by temporarily modifying their invisible and intangible properties. If you want this to be done differently for a certain overlay, give it an `isearch-open-invisible-temporary` property which is a function. The function is called with two arguments: the first is the overlay, and the second is `nil` to make the overlay visible, or `t` to make it invisible again.

## 11.7 Selective Display

*Selective display* refers to a pair of related features for hiding certain lines on the screen.

The first variant, explicit selective display, is designed for use in a Lisp program: it controls which lines are hidden by altering the text. This kind of hiding in some ways resembles the effect of the `invisible` property (see Section 11.6 [Invisible Text], page 121), but the two features are different and do not work the same way.

In the second variant, the choice of lines to hide is made automatically based on indentation. This variant is designed to be a user-level feature.

The way you control explicit selective display is by replacing a newline (control-j) with a carriage return (control-m). The text that was formerly a line following that newline is now hidden. Strictly speaking, it is temporarily no longer a line at all, since only newlines can separate lines; it is now part of the previous line.

Selective display does not directly affect editing commands. For example, `C-f` (`forward-char`) moves point unhesitatingly into hidden text. However, the replacement of newline characters with carriage return characters affects some editing commands. For example, `next-line` skips hidden lines, since it searches only for newlines. Modes that use selective display can also define commands that take account of the newlines, or that control which parts of the text are hidden.

When you write a selectively displayed buffer into a file, all the control-m's are output as newlines. This means that when you next read in the file, it looks OK, with nothing hidden. The selective display effect is seen only within Emacs.

`selective-display` [Variable]

This buffer-local variable enables selective display. This means that lines, or portions of lines, may be made hidden.

- If the value of `selective-display` is `t`, then the character control-m marks the start of hidden text; the control-m, and the rest of the line following it, are not displayed. This is explicit selective display.
- If the value of `selective-display` is a positive integer, then lines that start with more than that many columns of indentation are not displayed.

When some portion of a buffer is hidden, the vertical movement commands operate as if that portion did not exist, allowing a single `next-line` command to skip any number of hidden lines. However, character movement commands (such as `forward-char`) do not skip the hidden portion, and it is possible (if tricky) to insert or delete text in an hidden portion.

In the examples below, we show the *display appearance* of the buffer `foo`, which changes with the value of `selective-display`. The *contents* of the buffer do not change.

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------

(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

`selective-display-ellipses` [User Option]

If this buffer-local variable is non-`nil`, then Emacs displays '`...`' at the end of a line that is followed by hidden text. This example is a continuation of the previous one.

```
(setq selective-display-ellipses t)
     ⇒ t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

You can use a display table to substitute other text for the ellipsis ('`...`'). See Section 11.20.2 [Display Tables], page 190.

## 11.8 Temporary Displays

Temporary displays are used by Lisp programs to put output into a buffer and then present it to the user for perusal rather than for editing. Many help commands use this feature.

`with-output-to-temp-buffer` *buffer-name forms...* [Macro]

This function executes *forms* while arranging to insert any output they print into the buffer named *buffer-name*, which is first created if necessary, and put into Help mode. Finally, the buffer is displayed in some window, but not selected. (See the similar form `with-temp-buffer-window` below.)

If the *forms* do not change the major mode in the output buffer, so that it is still Help mode at the end of their execution, then `with-output-to-temp-buffer` makes this buffer read-only at the end, and also scans it for function and variable names to make them into clickable cross-references. See ⟨undefined⟩ [Tips for Documentation Strings], page ⟨undefined⟩, in particular the item on hyperlinks in documentation strings, for more details.

The string *buffer-name* specifies the temporary buffer, which need not already exist. The argument must be a string, not a buffer. The buffer is erased initially (with no questions asked), and it is marked as unmodified after `with-output-to-temp-buffer` exits.

`with-output-to-temp-buffer` binds `standard-output` to the temporary buffer, then it evaluates the forms in *forms*. Output using the Lisp output functions within *forms* goes by default to that buffer (but screen display and messages in the echo area, although they are "output" in the general sense of the word, are not affected). See ⟨undefined⟩ [Output Functions], page ⟨undefined⟩.

Several hooks are available for customizing the behavior of this construct; they are listed below.

The value of the last form in *forms* is returned.

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------
20

#<buffer foo>

---------- Buffer: foo ----------
```

`temp-buffer-show-function` [User Option]

If this variable is non-`nil`, `with-output-to-temp-buffer` calls it as a function to do the job of displaying a help buffer. The function gets one argument, which is the buffer it should display.

It is a good idea for this function to run `temp-buffer-show-hook` just as `with-output-to-temp-buffer` normally would, inside of `save-selected-window` and with the chosen window and buffer selected.

`temp-buffer-setup-hook` [Variable]

This normal hook is run by `with-output-to-temp-buffer` before evaluating *body*. When the hook runs, the temporary buffer is current. This hook is normally set up with a function to put the buffer in Help mode.

`temp-buffer-show-hook` [Variable]
This normal hook is run by `with-output-to-temp-buffer` after displaying the temporary buffer. When the hook runs, the temporary buffer is current, and the window it was displayed in is selected.

`with-temp-buffer-window` *buffer-or-name action quit-function forms...* [Macro]
This macro is similar to `with-output-to-temp-buffer`. Like that construct, it executes *forms* while arranging to insert any output they print into the buffer named *buffer-or-name*. Finally, the buffer is displayed in some window, but not selected. Unlike `with-output-to-temp-buffer`, this does not switch to Help mode.

The argument *buffer-or-name* specifies the temporary buffer. It can be either a buffer, which must already exist, or a string, in which case a buffer of that name is created if necessary. The buffer is marked as unmodified and read-only when `with-temp-buffer-window` exits.

This macro does not call `temp-buffer-show-function`. Rather, it passes the *action* argument to `display-buffer` in order to display the buffer.

The value of the last form in *forms* is returned, unless the argument *quit-function* is specified. In that case, it is called with two arguments: the window showing the buffer and the result of *forms*. The final return value is then whatever *quit-function* returns.

This macro uses the normal hooks `temp-buffer-window-setup-hook` and `temp-buffer-window-show-hook` in place of the analogous hooks run by `with-output-to-temp-buffer`.

`momentary-string-display` *string position* **&optional** *char message* [Function]
This function momentarily displays *string* in the current buffer at *position*. It has no effect on the undo list or on the buffer's modification status.

The momentary display remains until the next input event. If the next input event is *char*, `momentary-string-display` ignores it and returns. Otherwise, that event remains buffered for subsequent use as input. Thus, typing *char* will simply remove the string from the display, while typing (say) `C-f` will remove the string from the display and later (presumably) move point forward. The argument *char* is a space by default.

The return value of `momentary-string-display` is not meaningful.

If the string *string* does not contain control characters, you can do the same job in a more general way by creating (and then subsequently deleting) an overlay with a `before-string` property. See Section 11.9.2 [Overlay Properties], page 131.

If *message* is non-`nil`, it is displayed in the echo area while *string* is displayed in the buffer. If it is `nil`, a default message says to type *char* to continue.

In this example, point is initially located at the beginning of the second line:

```
---------- Buffer: foo ----------
This is the contents of foo.
*Second line.
---------- Buffer: foo ----------
```

```
(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

## 11.9 Overlays

You can use *overlays* to alter the appearance of a buffer's text on the screen, for the sake of presentation features. An overlay is an object that belongs to a particular buffer, and has a specified beginning and end. It also has properties that you can examine and set; these affect the display of the text within the overlay.

The visual effect of an overlay is the same as of the corresponding text property (see Section 22.19 [Text Properties], page 489). However, due to a different implementation, overlays generally don't scale well (many operations take a time that is proportional to the number of overlays in the buffer). If you need to affect the visual appearance of many portions in the buffer, we recommend using text properties.

An overlay uses markers to record its beginning and end; thus, editing the text of the buffer adjusts the beginning and end of each overlay so that it stays with the text. When you create the overlay, you can specify whether text inserted at the beginning should be inside the overlay or outside, and likewise for the end of the overlay.

### 11.9.1 Managing Overlays

This section describes the functions to create, delete and move overlays, and to examine their contents. Overlay changes are not recorded in the buffer's undo list, since the overlays are not part of the buffer's contents.

`overlayp` *object* [Function]
: This function returns `t` if *object* is an overlay.

`make-overlay` *start end* **&optional** *buffer front-advance rear-advance* [Function]
: This function creates and returns an overlay that belongs to *buffer* and ranges from *start* to *end*. Both *start* and *end* must specify buffer positions; they may be integers or markers. If *buffer* is omitted, the overlay is created in the current buffer.

  The arguments *front-advance* and *rear-advance* specify the marker insertion type for the start of the overlay and for the end of the overlay, respectively. See ⟨undefined⟩ [Marker Insertion Types], page ⟨undefined⟩. If they are both `nil`, the default, then the overlay extends to include any text inserted at the beginning, but not text inserted

at the end. If *front-advance* is non-`nil`, text inserted at the beginning of the overlay is excluded from the overlay. If *rear-advance* is non-`nil`, text inserted at the end of the overlay is included in the overlay.

`overlay-start` *overlay* [Function]
This function returns the position at which *overlay* starts, as an integer.

`overlay-end` *overlay* [Function]
This function returns the position at which *overlay* ends, as an integer.

`overlay-buffer` *overlay* [Function]
This function returns the buffer that *overlay* belongs to. It returns `nil` if *overlay* has been deleted.

`delete-overlay` *overlay* [Function]
This function deletes *overlay*. The overlay continues to exist as a Lisp object, and its property list is unchanged, but it ceases to be attached to the buffer it belonged to, and ceases to have any effect on display.

A deleted overlay is not permanently disconnected. You can give it a position in a buffer again by calling `move-overlay`.

`move-overlay` *overlay start end* **&optional** *buffer* [Function]
This function moves *overlay* to *buffer*, and places its bounds at *start* and *end*. Both arguments *start* and *end* must specify buffer positions; they may be integers or markers.

If *buffer* is omitted, *overlay* stays in the same buffer it was already associated with; if *overlay* was deleted, it goes into the current buffer.

The return value is *overlay*.

This is the only valid way to change the endpoints of an overlay. Do not try modifying the markers in the overlay by hand, as that fails to update other vital data structures and can cause some overlays to be "lost".

`remove-overlays` **&optional** *start end name value* [Function]
This function removes all the overlays between *start* and *end* whose property *name* has the value *value*. It can move the endpoints of the overlays in the region, or split them.

If *name* is omitted or `nil`, it means to delete all overlays in the specified region. If *start* and/or *end* are omitted or `nil`, that means the beginning and end of the buffer respectively. Therefore, `(remove-overlays)` removes all the overlays in the current buffer.

`copy-overlay` *overlay* [Function]
This function returns a copy of *overlay*. The copy has the same endpoints and properties as *overlay*. However, the marker insertion type for the start of the overlay and for the end of the overlay are set to their default values (see ⟨undefined⟩ [Marker Insertion Types], page ⟨undefined⟩).

Here are some examples:

```
;; Create an overlay.
(setq foo (make-overlay 1 10))
     ⇒ #<overlay from 1 to 10 in display.texi>
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 10
(overlay-buffer foo)
     ⇒ #<buffer display.texi>
;; Give it a property we can check later.
(overlay-put foo 'happy t)
     ⇒ t
;; Verify the property is present.
(overlay-get foo 'happy)
     ⇒ t
;; Move the overlay.
(move-overlay foo 5 20)
     ⇒ #<overlay from 5 to 20 in display.texi>
(overlay-start foo)
     ⇒ 5
(overlay-end foo)
     ⇒ 20
;; Delete the overlay.
(delete-overlay foo)
     ⇒ nil
;; Verify it is deleted.
foo
     ⇒ #<overlay in no buffer>
;; A deleted overlay has no position.
(overlay-start foo)
     ⇒ nil
(overlay-end foo)
     ⇒ nil
(overlay-buffer foo)
     ⇒ nil
;; Undelete the overlay.
(move-overlay foo 1 20)
     ⇒ #<overlay from 1 to 20 in display.texi>
;; Verify the results.
(overlay-start foo)
     ⇒ 1
(overlay-end foo)
     ⇒ 20
(overlay-buffer foo)
     ⇒ #<buffer display.texi>
```

```
;; Moving and deleting the overlay does not change its properties.
(overlay-get foo 'happy)
     ⇒ t
```

Emacs stores the overlays of each buffer in two lists, divided around an arbitrary "center position". One list extends backwards through the buffer from that center position, and the other extends forwards from that center position. The center position can be anywhere in the buffer.

`overlay-recenter` *pos* [Function]
: This function recenters the overlays of the current buffer around position *pos*. That makes overlay lookup faster for positions near *pos*, but slower for positions far away from *pos*.

A loop that scans the buffer forwards, creating overlays, can run faster if you do `(overlay-recenter (point-max))` first.

### 11.9.2 Overlay Properties

Overlay properties are like text properties in that the properties that alter how a character is displayed can come from either source. But in most respects they are different. See Section 22.19 [Text Properties], page 489, for comparison.

Text properties are considered a part of the text; overlays and their properties are specifically considered not to be part of the text. Thus, copying text between various buffers and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not. Unlike text property changes, overlay property changes are not recorded in the buffer's undo list.

Since more than one overlay can specify a property value for the same character, Emacs lets you specify a priority value of each overlay. You should not make assumptions about which overlay will prevail when there is a conflict and they have the same priority.

These functions read and set the properties of an overlay:

`overlay-get` *overlay prop* [Function]
: This function returns the value of property *prop* recorded in *overlay*, if any. If *overlay* does not record any value for that property, but it does have a `category` property which is a symbol, that symbol's *prop* property is used. Otherwise, the value is `nil`.

`overlay-put` *overlay prop value* [Function]
: This function sets the value of property *prop* recorded in *overlay* to *value*. It returns *value*.

`overlay-properties` *overlay* [Function]
: This returns a copy of the property list of *overlay*.

See also the function `get-char-property` which checks both overlay properties and text properties for a given character. See Section 22.19.1 [Examining Properties], page 489.

Many overlay properties have special meanings; here is a table of them:

`priority` This property's value (which should be a non-negative integer number) determines the priority of the overlay. No priority, or `nil`, means zero.

The priority matters when two or more overlays cover the same character and both specify the same property; the one whose `priority` value is larger overrides the other. For the `face` property, the higher priority overlay's value does not completely override the other value; instead, its face attributes override the face attributes of the lower priority `face` property.

Currently, all overlays take priority over text properties. Please avoid using negative priority values, as we have not yet decided just what they should mean.

`window` If the `window` property is non-`nil`, then the overlay applies only on that window.

`category` If an overlay has a `category` property, we call it the *category* of the overlay. It should be a symbol. The properties of the symbol serve as defaults for the properties of the overlay.

`face` This property controls the way text is displayed—for example, which font and which colors. See Section 11.12 [Faces], page 137, for more information.

In the simplest case, the value is a face name. It can also be a list; then each element can be any of these possibilities:

- A face name (a symbol or string).
- A property list of face attributes. This has the form (*keyword value* ...), where each *keyword* is a face attribute name and *value* is a meaningful value for that attribute. With this feature, you do not need to create a face each time you want to specify a particular attribute for certain text. See Section 11.12.1 [Face Attributes], page 138.
- A cons cell, of the form `(foreground-color . color-name)` or `(background-color . color-name)`. These elements specify just the foreground color or just the background color.

  `(foreground-color . color-name)` has the same effect as `(:foreground color-name)`; likewise for the background.

`mouse-face`
This property is used instead of `face` when the mouse is within the range of the overlay. However, Emacs ignores all face attributes from this property that alter the text size (e.g., `:height`, `:weight`, and `:slant`). Those attributes are always the same as in the unhighlighted text.

`display` This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrower, or replaced with an image. See Section 11.15 [Display Property], page 163.

`help-echo`
If an overlay has a `help-echo` property, then when you move the mouse onto the text in the overlay, Emacs displays a help string in the echo area, or in the tooltip window. For details see [Text help-echo], page 495.

`field` Consecutive characters with the same `field` property constitute a *field*. Some motion functions including `forward-word` and `beginning-of-line` stop moving at a field boundary. See Section 22.19.9 [Fields], page 504.

`modification-hooks`
This property's value is a list of functions to be called if any character within the overlay is changed or if text is inserted strictly within the overlay.

The hook functions are called both before and after each change. If the functions save the information they receive, and compare notes between calls, they can determine exactly what change has been made in the buffer text.

When called before a change, each function receives four arguments: the overlay, `nil`, and the beginning and end of the text range to be modified.

When called after a change, each function receives five arguments: the overlay, `t`, the beginning and end of the text range just modified, and the length of the pre-change text replaced by that range. (For an insertion, the pre-change length is zero; for a deletion, that length is the number of characters deleted, and the post-change beginning and end are equal.)

If these functions modify the buffer, they should bind `inhibit-modification-hooks` to `t` around doing so, to avoid confusing the internal mechanism that calls these hooks.

Text properties also support the `modification-hooks` property, but the details are somewhat different (see Section 22.19.4 [Special Properties], page 494).

`insert-in-front-hooks`
This property's value is a list of functions to be called before and after inserting text right at the beginning of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`insert-behind-hooks`
This property's value is a list of functions to be called before and after inserting text right at the end of the overlay. The calling conventions are the same as for the `modification-hooks` functions.

`invisible`
The `invisible` property can make the text in the overlay invisible, which means that it does not appear on the screen. See Section 11.6 [Invisible Text], page 121, for details.

`intangible`
The `intangible` property on an overlay works just like the `intangible` text property. See Section 22.19.4 [Special Properties], page 494, for details.

`isearch-open-invisible`
This property tells incremental search how to make an invisible overlay visible, permanently, if the final match overlaps it. See Section 11.6 [Invisible Text], page 121.

`isearch-open-invisible-temporary`
This property tells incremental search how to make an invisible overlay visible, temporarily, during the search. See Section 11.6 [Invisible Text], page 121.

`before-string`
: This property's value is a string to add to the display at the beginning of the overlay. The string does not appear in the buffer in any sense—only on the screen.

`after-string`
: This property's value is a string to add to the display at the end of the overlay. The string does not appear in the buffer in any sense—only on the screen.

`line-prefix`
: This property specifies a display spec to prepend to each non-continuation line at display-time. See Section 11.3 [Truncation], page 112.

`wrap-prefix`
: This property specifies a display spec to prepend to each continuation line at display-time. See Section 11.3 [Truncation], page 112.

`evaporate`
: If this property is non-`nil`, the overlay is deleted automatically if it becomes empty (i.e., if its length becomes zero). If you give an empty overlay a non-`nil` `evaporate` property, that deletes it immediately.

`local-map`
: If this property is non-`nil`, it specifies a keymap for a portion of the text. The property's value replaces the buffer's local map, when the character after point is within the overlay. See ⟨undefined⟩ [Active Keymaps], page ⟨undefined⟩.

`keymap`
: The `keymap` property is similar to `local-map` but overrides the buffer's local map (and the map specified by the `local-map` property) rather than replacing it.

The `local-map` and `keymap` properties do not affect a string displayed by the `before-string`, `after-string`, or `display` properties. This is only relevant for mouse clicks and other mouse events that fall on the string, since point is never on the string. To bind special mouse events for the string, assign it a `local-map` or `keymap` text property. See Section 22.19.4 [Special Properties], page 494.

### 11.9.3 Searching for Overlays

`overlays-at` *pos* [Function]
: This function returns a list of all the overlays that cover the character at position *pos* in the current buffer. The list is in no particular order. An overlay contains position *pos* if it begins at or before *pos*, and ends after *pos*.

To illustrate usage, here is a Lisp function that returns a list of the overlays that specify property *prop* for the character at point:

```
(defun find-overlays-specifying (prop)
  (let ((overlays (overlays-at (point)))
        found)
    (while overlays
      (let ((overlay (car overlays)))
        (if (overlay-get overlay prop)
            (setq found (cons overlay found))))
      (setq overlays (cdr overlays)))
    found))
```

`overlays-in` *beg end* [Function]
: This function returns a list of the overlays that overlap the region *beg* through *end*. "Overlap" means that at least one character is contained within the overlay and also contained within the specified region; however, empty overlays are included in the result if they are located at *beg*, strictly between *beg* and *end*, or at *end* when *end* denotes the position at the end of the buffer.

`next-overlay-change` *pos* [Function]
: This function returns the buffer position of the next beginning or end of an overlay, after *pos*. If there is none, it returns `(point-max)`.

`previous-overlay-change` *pos* [Function]
: This function returns the buffer position of the previous beginning or end of an overlay, before *pos*. If there is none, it returns `(point-min)`.

As an example, here's a simplified (and inefficient) version of the primitive function `next-single-char-property-change` (see Section 22.19.3 [Property Search], page 492). It searches forward from position *pos* for the next position where the value of a given property `prop`, as obtained from either overlays or text properties, changes.

```
(defun next-single-char-property-change (position prop)
  (save-excursion
    (goto-char position)
    (let ((propval (get-char-property (point) prop)))
      (while (and (not (eobp))
                  (eq (get-char-property (point) prop) propval))
        (goto-char (min (next-overlay-change (point))
                        (next-single-property-change (point) prop)))))
    (point)))
```

## 11.10 Width

Since not all characters have the same width, these functions let you check the width of a character. See Section 22.17.1 [Primitive Indent], page 483, and ⟨undefined⟩ [Screen Lines], page ⟨undefined⟩, for related functions.

`char-width` *char* [Function]
: This function returns the width in columns of the character *char*, if it were displayed in the current buffer (i.e., taking into account the buffer's display table, if any; see Section 11.20.2 [Display Tables], page 190). The width of a tab character is usually `tab-width` (see Section 11.20.1 [Usual Display], page 189).

`string-width` *string* [Function]
: This function returns the width in columns of the string *string*, if it were displayed in the current buffer and the selected window.

`truncate-string-to-width` *string width* **&optional** *start-column padding ellipsis* [Function]
: This function returns the part of *string* that fits within *width* columns, as a new string.

  If *string* does not reach *width*, then the result ends where *string* ends. If one multi-column character in *string* extends across the column *width*, that character is not

included in the result. Thus, the result can fall short of *width* but cannot go beyond it.

The optional argument *start-column* specifies the starting column. If this is non-`nil`, then the first *start-column* columns of the string are omitted from the value. If one multi-column character in *string* extends across the column *start-column*, that character is not included.

The optional argument *padding*, if non-`nil`, is a padding character added at the beginning and end of the result string, to extend it to exactly *width* columns. The padding character is used at the end of the result if it falls short of *width*. It is also used at the beginning of the result if one multi-column character in *string* extends across the column *start-column*.

If *ellipsis* is non-`nil`, it should be a string which will replace the end of *str* (including any padding) if it extends beyond *end-column*, unless the display width of *str* is equal to or less than the display width of *ellipsis*. If *ellipsis* is non-`nil` and not a string, it stands for `"..."`.

```
(truncate-string-to-width "\tab\t" 12 4)
     ⇒ "ab"
(truncate-string-to-width "\tab\t" 12 4 ?\s)
     ⇒ "    ab  "
```

## 11.11 Line Height

The total height of each display line consists of the height of the contents of the line, plus optional additional vertical line spacing above or below the display line.

The height of the line contents is the maximum height of any character or image on that display line, including the final newline if there is one. (A display line that is continued doesn't include a final newline.) That is the default line height, if you do nothing to specify a greater height. (In the most common case, this equals the height of the default frame font.)

There are several ways to explicitly specify a larger line height, either by specifying an absolute height for the display line, or by specifying vertical space. However, no matter what you specify, the actual line height can never be less than the default.

A newline can have a `line-height` text or overlay property that controls the total height of the display line ending in that newline.

If the property value is `t`, the newline character has no effect on the displayed height of the line—the visible contents alone determine the height. This is useful for tiling small images (or image slices) without adding blank areas between the images.

If the property value is a list of the form `(`*`height total`*`)`, that adds extra space *below* the display line. First Emacs uses *height* as a height spec to control extra space *above* the line; then it adds enough space *below* the line to bring the total line height up to *total*. In this case, the other ways to specify the line spacing are ignored.

Any other kind of property value is a height spec, which translates into a number—the specified line height. There are several ways to write a height spec; here's how each of them translates into a number:

*`integer`* If the height spec is a positive integer, the height value is that integer.

*float* If the height spec is a float, *float*, the numeric height value is *float* times the frame's default line height.

(*face* . *ratio*)
If the height spec is a cons of the format shown, the numeric height is *ratio* times the height of face *face*. *ratio* can be any type of number, or `nil` which means a ratio of 1. If *face* is `t`, it refers to the current face.

(`nil` . *ratio*)
If the height spec is a cons of the format shown, the numeric height is *ratio* times the height of the contents of the line.

Thus, any valid height spec determines the height in pixels, one way or another. If the line contents' height is less than that, Emacs adds extra vertical space above the line to achieve the specified total height.

If you don't specify the `line-height` property, the line's height consists of the contents' height plus the line spacing. There are several ways to specify the line spacing for different parts of Emacs text.

On graphical terminals, you can specify the line spacing for all lines in a frame, using the `line-spacing` frame parameter (see Section 18.3.3.4 [Layout Parameters], page 349). However, if the default value of `line-spacing` is non-`nil`, it overrides the frame's `line-spacing` parameter. An integer value specifies the number of pixels put below lines. A floating point number specifies the spacing relative to the frame's default line height.

You can specify the line spacing for all lines in a buffer via the buffer-local `line-spacing` variable. An integer value specifies the number of pixels put below lines. A floating point number specifies the spacing relative to the default frame line height. This overrides line spacings specified for the frame.

Finally, a newline can have a `line-spacing` text or overlay property that overrides the default frame line spacing and the buffer local `line-spacing` variable, for the display line ending in that newline.

One way or another, these mechanisms specify a Lisp value for the spacing of each line. The value is a height spec, and it translates into a Lisp value as described above. However, in this case the numeric height value specifies the line spacing, rather than the line height.

On text terminals, the line spacing cannot be altered.

## 11.12 Faces

A *face* is a collection of graphical *attributes* for displaying text: font, foreground color, background color, optional underlining, etc. Faces control how Emacs displays text in buffers, as well as other parts of the frame such as the mode line.

One way to represent a face is as a property list of attributes, like `(:foreground "red" :weight bold)`. For example, you can assign such an *anonymous face* as the value of the `face` text property; this causes Emacs to display the underlying text with the specified attributes. See Section 22.19.4 [Special Properties], page 494.

More commonly, a face is referred to via a *face name*: a Lisp symbol which is associated with a set of face attributes. Named faces are defined using the `defface` macro (see Section 11.12.2 [Defining Faces], page 141). Emacs defines several standard named faces; See Section "Standard Faces" in *The GNU Emacs Manual*.

Many parts of Emacs require named faces, and do not accept anonymous faces. These include the functions documented in Section 11.12.3 [Attribute Functions], page 143, and the variable `font-lock-keywords` (see Section 20.6.2 [Search-based Fontification], page 430). Unless otherwise stated, we will use the term *face* to refer only to named faces.

For backward compatibility, you can also use a string to specify a face name; that is equivalent to a Lisp symbol with the same name.

`facep` *object* [Function]
: This function returns a non-`nil` value if *object* is a named face: a Lisp symbol or string which serves as a face name. Otherwise, it returns `nil`.

By default, each face name corresponds to the same set of attributes in all frames. But you can also assign a face name a special set of attributes in one frame (see Section 11.12.3 [Attribute Functions], page 143).

### 11.12.1 Face Attributes

*Face attributes* determine the visual appearance of a face. The following table lists all the face attributes, their possible values, and their effects.

Apart from the values given below, each face attribute can have the value `unspecified`. This special value means that the face doesn't specify that attribute directly. An `unspecified` attribute tells Emacs to refer instead to a parent face (see the description `:inherit` attribute below); or, failing that, to an underlying face (see Section 11.12.4 [Displaying Faces], page 146). The `default` face must specify all attributes.

Some of these attributes are meaningful only on certain kinds of displays. If your display cannot handle a certain attribute, the attribute is ignored.

`:family`
: Font family or fontset (a string). See Section "Fonts" in *The GNU Emacs Manual*, for more information about font families. The function `font-family-list` (see below) returns a list of available family names. See Section 19.14 [Fontsets], page 389, for information about fontsets.

`:foundry`
: The name of the *font foundry* for the font family specified by the `:family` attribute (a string). See Section "Fonts" in *The GNU Emacs Manual*.

`:width`
: Relative character width. This should be one of the symbols `ultra-condensed`, `extra-condensed`, `condensed`, `semi-condensed`, `normal`, `semi-expanded`, `expanded`, `extra-expanded`, or `ultra-expanded`.

`:height`
: The height of the font. In the simplest case, this is an integer in units of 1/10 point.

  The value can also be a floating point number or a function, which specifies the height relative to an *underlying face* (see Section 11.12.4 [Displaying Faces], page 146). If the value is a floating point number, that specifies the amount by which to scale the height of the underlying face. If the value is a function, that function is called with one argument, the height of the underlying face, and returns the height of the new face. If the function is passed an integer argument, it must return an integer.

  The height of the default face must be specified using an integer; floating point and function values are not allowed.

`:weight` Font weight—one of the symbols (from densest to faintest) `ultra-bold`, `extra-bold`, `bold`, `semi-bold`, `normal`, `semi-light`, `light`, `extra-light`, or `ultra-light`. On text terminals which support variable-brightness text, any weight greater than normal is displayed as extra bright, and any weight less than normal is displayed as half-bright.

`:slant` Font slant—one of the symbols `italic`, `oblique`, `normal`, `reverse-italic`, or `reverse-oblique`. On text terminals that support variable-brightness text, slanted text is displayed as half-bright.

`:foreground`
Foreground color, a string. The value can be a system-defined color name, or a hexadecimal color specification. See Section 18.20 [Color Names], page 367. On black-and-white displays, certain shades of gray are implemented by stipple patterns.

`:background`
Background color, a string. The value can be a system-defined color name, or a hexadecimal color specification. See Section 18.20 [Color Names], page 367.

`:underline`
Whether or not characters should be underlined, and in what way. The possible values of the `:underline` attribute are:

`nil` Don't underline.

`t` Underline with the foreground color of the face.

*color* Underline in color *color*, a string specifying a color.

`(:color` *color* `:style` *style*`)`
*color* is either a string, or the symbol `foreground-color`, meaning the foreground color of the face. Omitting the attribute `:color` means to use the foreground color of the face. *style* should be a symbol `line` or `wave`, meaning to use a straight or wavy line. Omitting the attribute `:style` means to use a straight line.

`:overline`
Whether or not characters should be overlined, and in what color. If the value is `t`, overlining uses the foreground color of the face. If the value is a string, overlining uses that color. The value `nil` means do not overline.

`:strike-through`
Whether or not characters should be strike-through, and in what color. The value is used like that of `:overline`.

`:box` Whether or not a box should be drawn around characters, its color, the width of the box lines, and 3D appearance. Here are the possible values of the `:box` attribute, and what they mean:

`nil` Don't draw a box.

`t` Draw a box with lines of width 1, in the foreground color.

*color* Draw a box with lines of width 1, in color *color*.

`(:line-width` *width* `:color` *color* `:style` *style*`)`
: This way you can explicitly specify all aspects of the box. The value *width* specifies the width of the lines to draw; it defaults to 1. A negative width *-n* means to draw a line of width *n* that occupies the space of the underlying text, thus avoiding any increase in the character height or width.

  The value *color* specifies the color to draw with. The default is the foreground color of the face for simple boxes, and the background color of the face for 3D boxes.

  The value *style* specifies whether to draw a 3D box. If it is `released-button`, the box looks like a 3D button that is not being pressed. If it is `pressed-button`, the box looks like a 3D button that is being pressed. If it is `nil` or omitted, a plain 2D box is used.

`:inverse-video`
: Whether or not characters should be displayed in inverse video. The value should be `t` (yes) or `nil` (no).

`:stipple`
: The background stipple, a bitmap.

  The value can be a string; that should be the name of a file containing external-format X bitmap data. The file is found in the directories listed in the variable `x-bitmap-file-path`.

  Alternatively, the value can specify the bitmap directly, with a list of the form `(`*width height data*`)`. Here, *width* and *height* specify the size in pixels, and *data* is a string containing the raw bits of the bitmap, row by row. Each row occupies (*width*+7)/8 consecutive bytes in the string (which should be a unibyte string for best results). This means that each row always occupies at least one whole byte.

  If the value is `nil`, that means use no stipple pattern.

  Normally you do not need to set the stipple attribute, because it is used automatically to handle certain shades of gray.

`:font`
: The font used to display the face. Its value should be a font object. See Section 11.12.9 [Font Selection], page 150, for information about font objects.

  When specifying this attribute using `set-face-attribute` (see Section 11.12.3 [Attribute Functions], page 143), you may also supply a font spec, a font entity, or a string. Emacs converts such values to an appropriate font object, and stores that font object as the actual attribute value. If you specify a string, the contents of the string should be a font name (see Section "Fonts" in *The GNU Emacs Manual*); if the font name is an XLFD containing wildcards, Emacs chooses the first font matching those wildcards. Specifying this attribute also changes the values of the `:family`, `:foundry`, `:width`, `:height`, `:weight`, and `:slant` attributes.

`:inherit`
: The name of a face from which to inherit attributes, or a list of face names. Attributes from inherited faces are merged into the face like an underlying face would be, with higher priority than underlying faces (see Section 11.12.4

[Displaying Faces], page 146). If a list of faces is used, attributes from faces earlier in the list override those from later faces.

`font-family-list` **&optional** *frame* [Function]
This function returns a list of available font family names. The optional argument *frame* specifies the frame on which the text is to be displayed; if it is `nil`, the selected frame is used.

`underline-minimum-offset` [User Option]
This variable specifies the minimum distance between the baseline and the underline, in pixels, when displaying underlined text.

`x-bitmap-file-path` [User Option]
This variable specifies a list of directories for searching for bitmap files, for the `:stipple` attribute.

`bitmap-spec-p` *object* [Function]
This returns `t` if *object* is a valid bitmap specification, suitable for use with `:stipple` (see above). It returns `nil` otherwise.

### 11.12.2 Defining Faces

The usual way to define a face is through the `defface` macro. This macro defines a face name, and associates that name with a set of face attributes. It also sets up the face so that the user can customize it via the Customize interface (see Chapter 33 [Customization], page 686).

`defface` *face spec doc* [*keyword value*]... [Macro]
This macro declares *face* as a customizable face whose default attributes are given by *spec*. You should not quote the symbol *face*, and it should not end in '`-face`' (that would be redundant). The argument *doc* is a documentation string for the face. The additional *keyword* arguments have the same meanings as in `defgroup` and `defcustom` (see ⟨undefined⟩ [Common Keywords], page ⟨undefined⟩).

When `defface` executes, it defines the face according to *spec*, then uses any customizations that were read from the init file (see Section 33.4 [Init File], page 711) to override that specification.

When you evaluate a `defface` form with `C-M-x` in Emacs Lisp mode (`eval-defun`), a special feature of `eval-defun` overrides any customizations of the face. This way, the face reflects exactly what the `defface` says.

The *spec* argument is a *face specification*, which states how the face should appear on different kinds of terminals. It should be an alist whose elements each have the form

```
(display . plist)
```

*display* specifies a class of terminals (see below). *plist* is a property list of face attributes and their values, specifying how the face appears on such terminals. For backward compatibility, you can also write an element as `(display plist)`.

The *display* part of an element of *spec* determines which terminals the element matches. If more than one element of *spec* matches a given terminal, the first element that matches is the one used for that terminal. There are three possibilities for *display*:

`default` This element of *spec* doesn't match any terminal; instead, it specifies defaults that apply to all terminals. This element, if used, must be the first element of *spec*. Each of the following elements can override any or all of these defaults.

`t` This element of *spec* matches all terminals. Therefore, any subsequent elements of *spec* are never used. Normally `t` is used in the last (or only) element of *spec*.

a list If *display* is a list, each element should have the form `(`*`characteristic value...`*`)`. Here *characteristic* specifies a way of classifying terminals, and the *value*s are possible classifications which *display* should apply to. Here are the possible values of *characteristic*:

`type` The kind of window system the terminal uses—either `graphic` (any graphics-capable display), `x`, `pc` (for the MS-DOS console), `w32` (for MS Windows 9X/NT/2K/XP), or `tty` (a non-graphics-capable display). See Section 11.22 [Window Systems], page 194.

`class` What kinds of colors the terminal supports—either `color`, `grayscale`, or `mono`.

`background`
The kind of background—either `light` or `dark`.

`min-colors`
An integer that represents the minimum number of colors the terminal should support. This matches a terminal if its `display-color-cells` value is at least the specified integer.

`supports` Whether or not the terminal can display the face attributes given in *value*... (see Section 11.12.1 [Face Attributes], page 138). See [Display Face Attribute Testing], page 371, for more information on exactly how this testing is done.

If an element of *display* specifies more than one *value* for a given *characteristic*, any of those values is acceptable. If *display* has more than one element, each element should specify a different *characteristic*; then *each* characteristic of the terminal must match one of the *value*s specified for it in *display*.

Here's how the standard face `highlight` is defined:

```
(defface highlight
  '((((class color) (min-colors 88) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 88) (background dark))
```

```
     :background "darkolivegreen")
    (((class color) (min-colors 16) (background light))
     :background "darkseagreen2")
    (((class color) (min-colors 16) (background dark))
     :background "darkolivegreen")
    (((class color) (min-colors 8))
     :background "green" :foreground "black")
    (t :inverse-video t))
  "Basic face for highlighting."
  :group 'basic-faces)
```

Internally, Emacs stores the face's default specification in its `face-defface-spec` symbol property (see ⟨undefined⟩ [Symbol Properties], page ⟨undefined⟩). The `saved-face` property stores the face specification saved by the user, using the customization buffer; the `customized-face` property stores the face specification customized for the current session, but not saved; and the `theme-face` property stores an alist associating the active customization settings and Custom themes with their specifications for that face. The face's documentation string is stored in the `face-documentation` property. But normally you should not try to set any of these properties directly. See ⟨undefined⟩ [Applying Customizations], page ⟨undefined⟩, for the `custom-set-faces` function, which is used to apply customized face settings.

People are sometimes tempted to create variables whose values specify a face to use. In the vast majority of cases, this is not necessary; it is preferable to simply use faces directly.

### 11.12.3 Face Attribute Functions

This section describes the functions for accessing and modifying the attributes of an existing named face.

`set-face-attribute` *face frame* **&rest** *arguments* [Function]

This function sets one or more attributes of *face* for *frame*. The attributes you specify this way override whatever the `defface` says.

The extra arguments *arguments* specify the attributes to set, and the values for them. They should consist of alternating attribute names (such as `:family` or `:underline`) and values. Thus,

```
(set-face-attribute 'foo nil
                    :width 'extended
                    :weight 'bold)
```

sets the attribute `:width` to `extended` and the attribute `:weight` to `bold`.

If *frame* is `t`, this function sets the default attributes for new frames. Default attribute values specified this way override the `defface` for newly created frames.

If *frame* is `nil`, this function sets the attributes for all existing frames, and the default for new frames.

`face-attribute` *face attribute* **&optional** *frame inherit* [Function]

This returns the value of the *attribute* attribute of *face* on *frame*. If *frame* is `nil`, that means the selected frame (see Section 18.9 [Input Focus], page 358).

If *frame* is `t`, this returns whatever new-frames default value you previously specified with `set-face-attribute` for the *attribute* attribute of *face*. If you have not specified one, it returns `nil`.

If *inherit* is `nil`, only attributes directly defined by *face* are considered, so the return value may be `unspecified`, or a relative value. If *inherit* is non-`nil`, *face*'s definition of *attribute* is merged with the faces specified by its `:inherit` attribute; however the return value may still be `unspecified` or relative. If *inherit* is a face or a list of faces, then the result is further merged with that face (or faces), until it becomes specified and absolute.

To ensure that the return value is always specified and absolute, use a value of `default` for *inherit*; this will resolve any unspecified or relative values by merging with the `default` face (which is always completely specified).

For example,

```
(face-attribute 'bold :weight)
     ⇒ bold
```

`face-attribute-relative-p` *attribute value* [Function]

This function returns non-`nil` if *value*, when used as the value of the face attribute *attribute*, is relative. This means it would modify, rather than completely override, any value that comes from a subsequent face in the face list or that is inherited from another face.

`unspecified` is a relative value for all attributes. For `:height`, floating point and function values are also relative.

For example:

```
(face-attribute-relative-p :height 2.0)
     ⇒ t
```

`face-all-attributes` *face* **&optional** *frame* [Function]

This function returns an alist of attributes of *face*. The elements of the result are name-value pairs of the form `(attr-name . attr-value)`. Optional argument *frame* specifies the frame whose definition of *face* to return; if omitted or `nil`, the returned value describes the default attributes of *face* for newly created frames.

`merge-face-attribute` *attribute value1 value2* [Function]

If *value1* is a relative value for the face attribute *attribute*, returns it merged with the underlying value *value2*; otherwise, if *value1* is an absolute value for the face attribute *attribute*, returns *value1* unchanged.

The following commands and functions mostly provide compatibility with old versions of Emacs. They work by calling `set-face-attribute`. Values of `t` and `nil` for their *frame* argument are handled just like `set-face-attribute` and `face-attribute`. The commands read their arguments using the minibuffer, if called interactively.

`set-face-foreground` *face color* **&optional** *frame* [Command]
`set-face-background` *face color* **&optional** *frame* [Command]

These set the `:foreground` attribute (or `:background` attribute, respectively) of *face* to *color*.

`set-face-stipple` *face pattern* **&optional** *frame* [Command]
This sets the `:stipple` attribute of *face* to *pattern*.

`set-face-font` *face font* **&optional** *frame* [Command]
This sets the `:font` attribute of *face* to *font*.

`set-face-bold-p` *face bold-p* **&optional** *frame* [Function]
This sets the `:weight` attribute of *face* to *normal* if *bold-p* is `nil`, and to *bold* otherwise.

`set-face-italic-p` *face italic-p* **&optional** *frame* [Function]
This sets the `:slant` attribute of *face* to *normal* if *italic-p* is `nil`, and to *italic* otherwise.

`set-face-underline` *face underline* **&optional** *frame* [Function]
This sets the `:underline` attribute of *face* to *underline*.

`set-face-inverse-video-p` *face inverse-video-p* **&optional** *frame* [Function]
This sets the `:inverse-video` attribute of *face* to *inverse-video-p*.

`invert-face` *face* **&optional** *frame* [Command]
This swaps the foreground and background colors of face *face*.

The following functions examine the attributes of a face. If you don't specify *frame*, they refer to the selected frame; `t` refers to the default data for new frames. They return the symbol `unspecified` if the face doesn't define any value for that attribute.

`face-foreground` *face* **&optional** *frame inherit* [Function]
`face-background` *face* **&optional** *frame inherit* [Function]
These functions return the foreground color (or background color, respectively) of face *face*, as a string.

If *inherit* is `nil`, only a color directly defined by the face is returned. If *inherit* is non-`nil`, any faces specified by its `:inherit` attribute are considered as well, and if *inherit* is a face or a list of faces, then they are also considered, until a specified color is found. To ensure that the return value is always specified, use a value of `default` for *inherit*.

`face-stipple` *face* **&optional** *frame inherit* [Function]
This function returns the name of the background stipple pattern of face *face*, or `nil` if it doesn't have one.

If *inherit* is `nil`, only a stipple directly defined by the face is returned. If *inherit* is non-`nil`, any faces specified by its `:inherit` attribute are considered as well, and if *inherit* is a face or a list of faces, then they are also considered, until a specified stipple is found. To ensure that the return value is always specified, use a value of `default` for *inherit*.

`face-font` *face* **&optional** *frame* [Function]
This function returns the name of the font of face *face*.

`face-bold-p` *face* **&optional** *frame* [Function]
: This function returns a non-`nil` value if the `:weight` attribute of *face* is bolder than normal (i.e., one of `semi-bold`, `bold`, `extra-bold`, or `ultra-bold`). Otherwise, it returns `nil`.

`face-italic-p` *face* **&optional** *frame* [Function]
: This function returns a non-`nil` value if the `:slant` attribute of *face* is `italic` or `oblique`, and `nil` otherwise.

`face-underline-p` *face* **&optional** *frame* [Function]
: This function returns non-`nil` if face *face* specifies a non-`nil` `:underline` attribute.

`face-inverse-video-p` *face* **&optional** *frame* [Function]
: This function returns non-`nil` if face *face* specifies a non-`nil` `:inverse-video` attribute.

### 11.12.4 Displaying Faces

When Emacs displays a given piece of text, the visual appearance of the text may be determined by faces drawn from different sources. If these various sources together specify more than one face for a particular character, Emacs merges the attributes of the various faces. Here is the order in which Emacs merges the faces, from highest to lowest priority:

- If the text consists of a special glyph, the glyph can specify a particular face. See Section 11.20.4 [Glyphs], page 192.
- If the text lies within an active region, Emacs highlights it using the `region` face. See Section "Standard Faces" in *The GNU Emacs Manual*.
- If the text lies within an overlay with a non-`nil` `face` property, Emacs applies the face(s) specified by that property. If the overlay has a `mouse-face` property and the mouse is "near enough" to the overlay, Emacs applies the face or face attributes specified by the `mouse-face` property instead. See Section 11.9.2 [Overlay Properties], page 131.

  When multiple overlays cover one character, an overlay with higher priority overrides those with lower priority. See Section 11.9 [Overlays], page 128.
- If the text contains a `face` or `mouse-face` property, Emacs applies the specified faces and face attributes. See Section 22.19.4 [Special Properties], page 494. (This is how Font Lock mode faces are applied. See Section 20.6 [Font Lock Mode], page 429.)
- If the text lies within the mode line of the selected window, Emacs applies the `mode-line` face. For the mode line of a non-selected window, Emacs applies the `mode-line-inactive` face. For a header line, Emacs applies the `header-line` face.
- If any given attribute has not been specified during the preceding steps, Emacs applies the attribute of the `default` face.

At each stage, if a face has a valid `:inherit` attribute, Emacs treats any attribute with an `unspecified` value as having the corresponding value drawn from the parent face(s). see Section 11.12.1 [Face Attributes], page 138. Note that the parent face(s) may also leave the attribute unspecified; in that case, the attribute remains unspecified at the next level of face merging.

### 11.12.5 Face Remapping

The variable `face-remapping-alist` is used for buffer-local or global changes in the appearance of a face. For instance, it is used to implement the `text-scale-adjust` command (see Section "Text Scale" in *The GNU Emacs Manual*).

`face-remapping-alist` [Variable]

The value of this variable is an alist whose elements have the form `(face . remapping)`. This causes Emacs to display any text having the face *face* with *remapping*, rather than the ordinary definition of *face*.

*remapping* may be any face specification suitable for a `face` text property: either a face (i.e., a face name or a property list of attribute/value pairs), or a list of faces. For details, see the description of the `face` text property in Section 22.19.4 [Special Properties], page 494. *remapping* serves as the complete specification for the remapped face—it replaces the normal definition of *face*, instead of modifying it.

If `face-remapping-alist` is buffer-local, its local value takes effect only within that buffer.

Note: face remapping is non-recursive. If *remapping* references the same face name *face*, either directly or via the `:inherit` attribute of some other face in *remapping*, that reference uses the normal definition of *face*. For instance, if the `mode-line` face is remapped using this entry in `face-remapping-alist`:

```
(mode-line italic mode-line)
```

then the new definition of the `mode-line` face inherits from the `italic` face, and the *normal* (non-remapped) definition of `mode-line` face.

The following functions implement a higher-level interface to `face-remapping-alist`. Most Lisp code should use these functions instead of setting `face-remapping-alist` directly, to avoid trampling on remappings applied elsewhere. These functions are intended for buffer-local remappings, so they all make `face-remapping-alist` buffer-local as a side-effect. They manage `face-remapping-alist` entries of the form

```
(face relative-spec-1 relative-spec-2 ... base-spec)
```

where, as explained above, each of the *relative-spec-N* and *base-spec* is either a face name, or a property list of attribute/value pairs. Each of the *relative remapping* entries, *relative-spec-N*, is managed by the `face-remap-add-relative` and `face-remap-remove-relative` functions; these are intended for simple modifications like changing the text size. The *base remapping* entry, *base-spec*, has the lowest priority and is managed by the `face-remap-set-base` and `face-remap-reset-base` functions; it is intended for major modes to remap faces in the buffers they control.

`face-remap-add-relative` *face* **&rest** *specs* [Function]

This functions adds the face specifications in *specs* as relative remappings for face *face* in the current buffer. The remaining arguments, *specs*, should form either a list of face names, or a property list of attribute/value pairs.

The return value is a Lisp object that serves as a "cookie"; you can pass this object as an argument to `face-remap-remove-relative` if you need to remove the remapping later.

```
;; Remap the 'escape-glyph' face into a combination
;; of the 'highlight' and 'italic' faces:
(face-remap-add-relative 'escape-glyph 'highlight 'italic)

;; Increase the size of the 'default' face by 50%:
(face-remap-add-relative 'default :height 1.5)
```

`face-remap-remove-relative` *cookie* [Function]
This function removes a relative remapping previously added by `face-remap-add-relative`. *cookie* should be the Lisp object returned by `face-remap-add-relative` when the remapping was added.

`face-remap-set-base` *face* **&rest** *specs* [Function]
This function sets the base remapping of *face* in the current buffer to *specs*. If *specs* is empty, the default base remapping is restored, similar to calling `face-remap-reset-base` (see below); note that this is different from *specs* containing a single value `nil`, which has the opposite result (the global definition of *face* is ignored).

This overwrites the default *base-spec*, which inherits the global face definition, so it is up to the caller to add such inheritance if so desired.

`face-remap-reset-base` *face* [Function]
This function sets the base remapping of *face* to its default value, which inherits from *face*'s global definition.

### 11.12.6 Functions for Working with Faces

Here are additional functions for creating and working with faces.

`face-list` [Function]
This function returns a list of all defined face names.

`face-id` *face* [Function]
This function returns the *face number* of face *face*. This is a number that uniquely identifies a face at low levels within Emacs. It is seldom necessary to refer to a face by its face number.

`face-documentation` *face* [Function]
This function returns the documentation string of face *face*, or `nil` if none was specified for it.

`face-equal` *face1 face2* **&optional** *frame* [Function]
This returns `t` if the faces *face1* and *face2* have the same attributes for display.

`face-differs-from-default-p` *face* **&optional** *frame* [Function]
This returns non-`nil` if the face *face* displays differently from the default face.

A *face alias* provides an equivalent name for a face. You can define a face alias by giving the alias symbol the `face-alias` property, with a value of the target face name. The following example makes `modeline` an alias for the `mode-line` face.

```
(put 'modeline 'face-alias 'mode-line)
```

`define-obsolete-face-alias` *obsolete-face current-face when* [Macro]
This macro defines `obsolete-face` as an alias for *current-face*, and also marks it as obsolete, indicating that it may be removed in future. *when* should be a string indicating when `obsolete-face` was made obsolete (usually a version number string).

### 11.12.7 Automatic Face Assignment

This hook is used for automatically assigning faces to text in the buffer. It is part of the implementation of Jit-Lock mode, used by Font-Lock.

`fontification-functions` [Variable]
This variable holds a list of functions that are called by Emacs redisplay as needed, just before doing redisplay. They are called even when Font Lock Mode isn't enabled. When Font Lock Mode is enabled, this variable usually holds just one function, `jit-lock-function`.

The functions are called in the order listed, with one argument, a buffer position *pos*. Collectively they should attempt to assign faces to the text in the current buffer starting at *pos*.

The functions should record the faces they assign by setting the `face` property. They should also add a non-`nil` `fontified` property to all the text they have assigned faces to. That property tells redisplay that faces have been assigned to that text already.

It is probably a good idea for the functions to do nothing if the character after *pos* already has a non-`nil` `fontified` property, but this is not required. If one function overrides the assignments made by a previous one, the properties after the last function finishes are the ones that really matter.

For efficiency, we recommend writing these functions so that they usually assign faces to around 400 to 600 characters at each call.

### 11.12.8 Basic Faces

If your Emacs Lisp program needs to assign some faces to text, it is often a good idea to use certain existing faces or inherit from them, rather than defining entirely new faces. This way, if other users have customized the basic faces to give Emacs a certain look, your program will "fit in" without additional customization.

Some of the basic faces defined in Emacs are listed below. In addition to these, you might want to make use of the Font Lock faces for syntactic highlighting, if highlighting is not already handled by Font Lock mode, or if some Font Lock faces are not in use. See Section 20.6.7 [Faces for Font Lock], page 436.

`default` The default face, whose attributes are all specified. All other faces implicitly inherit from it: any unspecified attribute defaults to the attribute on this face (see Section 11.12.1 [Face Attributes], page 138).

`bold`
`italic`
`bold-italic`
`underline`
`fixed-pitch`
`variable-pitch`
: These have the attributes indicated by their names (e.g., `bold` has a bold `:weight` attribute), with all other attributes unspecified (and so given by `default`).

`shadow`
: For "dimmed out" text. For example, it is used for the ignored part of a filename in the minibuffer (see Section "Minibuffers for File Names" in *The GNU Emacs Manual*).

`link`
`link-visited`
: For clickable text buttons that send the user to a different buffer or "location".

`highlight`
: For stretches of text that should temporarily stand out. For example, it is commonly assigned to the `mouse-face` property for cursor highlighting (see Section 22.19.4 [Special Properties], page 494).

`match`
: For text matching a search command.

`error`
`warning`
`success`
: For text concerning errors, warnings, or successes. For example, these are used for messages in '`*Compilation*`' buffers.

### 11.12.9 Font Selection

Before Emacs can draw a character on a graphical display, it must select a *font* for that character[1]. See Section "Fonts" in *The GNU Emacs Manual*. Normally, Emacs automatically chooses a font based on the faces assigned to that character—specifically, the face attributes `:family`, `:weight`, `:slant`, and `:width` (see Section 11.12.1 [Face Attributes], page 138). The choice of font also depends on the character to be displayed; some fonts can only display a limited set of characters. If no available font exactly fits the requirements, Emacs looks for the *closest matching font*. The variables in this section control how Emacs makes this selection.

`face-font-family-alternatives` [User Option]
: If a given family is specified but does not exist, this variable specifies alternative font families to try. Each element should have this form:

```
(family alternate-families...)
```

: If *family* is specified but not available, Emacs will try the other families given in *alternate-families*, one by one, until it finds a family that does exist.

[1] In this context, the term *font* has nothing to do with Font Lock (see Section 20.6 [Font Lock Mode], page 429).

`face-font-selection-order` [User Option]

If there is no font that exactly matches all desired face attributes (`:width`, `:height`, `:weight`, and `:slant`), this variable specifies the order in which these attributes should be considered when selecting the closest matching font. The value should be a list containing those four attribute symbols, in order of decreasing importance. The default is `(:width :height :weight :slant)`.

Font selection first finds the best available matches for the first attribute in the list; then, among the fonts which are best in that way, it searches for the best matches in the second attribute, and so on.

The attributes `:weight` and `:width` have symbolic values in a range centered around `normal`. Matches that are more extreme (farther from `normal`) are somewhat preferred to matches that are less extreme (closer to `normal`); this is designed to ensure that non-normal faces contrast with normal ones, whenever possible.

One example of a case where this variable makes a difference is when the default font has no italic equivalent. With the default ordering, the `italic` face will use a non-italic font that is similar to the default one. But if you put `:slant` before `:height`, the `italic` face will use an italic font, even if its height is not quite right.

`face-font-registry-alternatives` [User Option]

This variable lets you specify alternative font registries to try, if a given registry is specified and doesn't exist. Each element should have this form:

```
(registry alternate-registries...)
```

If *registry* is specified but not available, Emacs will try the other registries given in *alternate-registries*, one by one, until it finds a registry that does exist.

Emacs can make use of scalable fonts, but by default it does not use them.

`scalable-fonts-allowed` [User Option]

This variable controls which scalable fonts to use. A value of `nil`, the default, means do not use scalable fonts. `t` means to use any scalable font that seems appropriate for the text.

Otherwise, the value must be a list of regular expressions. Then a scalable font is enabled for use if its name matches any regular expression in the list. For example,

```
(setq scalable-fonts-allowed '("muleindian-2$"))
```

allows the use of scalable fonts with registry `muleindian-2`.

`face-font-rescale-alist` [Variable]

This variable specifies scaling for certain faces. Its value should be a list of elements of the form

```
(fontname-regexp . scale-factor)
```

If *fontname-regexp* matches the font name that is about to be used, this says to choose a larger similar font according to the factor *scale-factor*. You would use this feature to normalize the font size if certain fonts are bigger or smaller than their nominal heights and widths would suggest.

### 11.12.10 Looking Up Fonts

`x-list-fonts` *name* **&optional** *reference-face frame maximum width* [Function]

This function returns a list of available font names that match *name*. *name* should be a string containing a font name in either the Fontconfig, GTK, or XLFD format (see Section "Fonts" in *The GNU Emacs Manual*). Within an XLFD string, wildcard characters may be used: the '`*`' character matches any substring, and the '`?`' character matches any single character. Case is ignored when matching font names.

If the optional arguments *reference-face* and *frame* are specified, the returned list includes only fonts that are the same size as *reference-face* (a face name) currently is on the frame *frame*.

The optional argument *maximum* sets a limit on how many fonts to return. If it is non-`nil`, then the return value is truncated after the first *maximum* matching fonts. Specifying a small value for *maximum* can make this function much faster, in cases where many fonts match the pattern.

The optional argument *width* specifies a desired font width. If it is non-`nil`, the function only returns those fonts whose characters are (on average) *width* times as wide as *reference-face*.

`x-family-fonts` **&optional** *family frame* [Function]

This function returns a list describing the available fonts for family *family* on *frame*. If *family* is omitted or `nil`, this list applies to all families, and therefore, it contains all available fonts. Otherwise, *family* must be a string; it may contain the wildcards '`?`' and '`*`'.

The list describes the display that *frame* is on; if *frame* is omitted or `nil`, it applies to the selected frame's display (see Section 18.9 [Input Focus], page 358).

Each element in the list is a vector of the following form:

```
[family width point-size weight slant
 fixed-p full registry-and-encoding]
```

The first five elements correspond to face attributes; if you specify these attributes for a face, it will use this font.

The last three elements give additional information about the font. *fixed-p* is non-`nil` if the font is fixed-pitch. *full* is the full name of the font, and *registry-and-encoding* is a string giving the registry and encoding of the font.

### 11.12.11 Fontsets

A *fontset* is a list of fonts, each assigned to a range of character codes. An individual font cannot display the whole range of characters that Emacs supports, but a fontset can. Fontsets have names, just as fonts do, and you can use a fontset name in place of a font name when you specify the "font" for a frame or a face. Here is information about defining a fontset under Lisp program control.

`create-fontset-from-fontset-spec` *fontset-spec* **&optional** *style-variant-p noerror* [Function]

This function defines a new fontset according to the specification string *fontset-spec*. The string should have this format:

*fontpattern*, [*charset*:*font*]...

Whitespace characters before and after the commas are ignored.

The first part of the string, *fontpattern*, should have the form of a standard X font name, except that the last two fields should be '`fontset-alias`'.

The new fontset has two names, one long and one short. The long name is *fontpattern* in its entirety. The short name is '`fontset-alias`'. You can refer to the fontset by either name. If a fontset with the same name already exists, an error is signaled, unless *noerror* is non-`nil`, in which case this function does nothing.

If optional argument *style-variant-p* is non-`nil`, that says to create bold, italic and bold-italic variants of the fontset as well. These variant fontsets do not have a short name, only a long one, which is made by altering *fontpattern* to indicate the bold or italic status.

The specification string also says which fonts to use in the fontset. See below for the details.

The construct '*charset*:*font*' specifies which font to use (in this fontset) for one particular character set. Here, *charset* is the name of a character set, and *font* is the font to use for that character set. You can use this construct any number of times in the specification string.

For the remaining character sets, those that you don't specify explicitly, Emacs chooses a font based on *fontpattern*: it replaces '`fontset-alias`' with a value that names one character set. For the ASCII character set, '`fontset-alias`' is replaced with '`ISO8859-1`'.

In addition, when several consecutive fields are wildcards, Emacs collapses them into a single wildcard. This is to prevent use of auto-scaled fonts. Fonts made by scaling larger fonts are not usable for editing, and scaling a smaller font is not useful because it is better to use the smaller font in its own size, which Emacs does.

Thus if *fontpattern* is this,

```
-*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24
```

the font specification for ASCII characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-ISO8859-1
```

and the font specification for Chinese GB2312 characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-gb2312*-*
```

You may not have any Chinese font matching the above font specification. Most X distributions include only Chinese fonts that have '`song ti`' or '`fangsong ti`' in the *family* field. In such a case, '`Fontset-n`' can be specified as below:

```
Emacs.Fontset-0: -*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24,\
        chinese-gb2312:-*-*-medium-r-normal-*-24-*-gb2312*-*
```

Then, the font specifications for all but Chinese GB2312 characters have '`fixed`' in the *family* field, and the font specification for Chinese GB2312 characters has a wild card '`*`' in the *family* field.

`set-fontset-font` *name character font-spec* **&optional** *frame add* [Function]

This function modifies the existing fontset *name* to use the font matching with *font-spec* for the character *character*.

If *name* is `nil`, this function modifies the fontset of the selected frame or that of *frame* if *frame* is not `nil`.

If *name* is `t`, this function modifies the default fontset, whose short name is '`fontset-default`'.

*character* may be a cons; `(from . to)`, where *from* and *to* are character codepoints. In that case, use *font-spec* for all characters in the range *from* and *to* (inclusive).

*character* may be a charset. In that case, use *font-spec* for all character in the charsets.

*character* may be a script name. In that case, use *font-spec* for all character in the charsets.

*font-spec* may be a cons; `(family . registry)`, where *family* is a family name of a font (possibly including a foundry name at the head), *registry* is a registry name of a font (possibly including an encoding name at the tail).

*font-spec* may be a font name string.

The optional argument *add*, if non-`nil`, specifies how to add *font-spec* to the font specifications previously set. If it is `prepend`, *font-spec* is prepended. If it is `append`, *font-spec* is appended. By default, *font-spec* overrides the previous settings.

For instance, this changes the default fontset to use a font of which family name is '`Kochi Gothic`' for all characters belonging to the charset `japanese-jisx0208`.

```
(set-fontset-font t 'japanese-jisx0208
                  (font-spec :family "Kochi Gothic"))
```

`char-displayable-p` *char* [Function]

This function returns `t` if Emacs ought to be able to display *char*. More precisely, if the selected frame's fontset has a font to display the character set that *char* belongs to.

Fontsets can specify a font on a per-character basis; when the fontset does that, this function's value may not be accurate.

### 11.12.12 Low-Level Font Representation

Normally, it is not necessary to manipulate fonts directly. In case you need to do so, this section explains how.

In Emacs Lisp, fonts are represented using three different Lisp object types: *font objects*, *font specs*, and *font entities*.

`fontp` *object* **&optional** *type* [Function]

Return `t` if *object* is a font object, font spec, or font entity. Otherwise, return `nil`.

The optional argument *type*, if non-`nil`, determines the exact type of Lisp object to check for. In that case, *type* should be one of `font-object`, `font-spec`, or `font-entity`.

A font object is a Lisp object that represents a font that Emacs has *opened*. Font objects cannot be modified in Lisp, but they can be inspected.

`font-at` *position* **&optional** *window string* [Function]

Return the font object that is being used to display the character at position *position* in the window *window*. If *window* is `nil`, it defaults to the selected window. If *string*

is `nil`, *position* specifies a position in the current buffer; otherwise, *string* should be a string, and *position* specifies a position in that string.

A font spec is a Lisp object that contains a set of specifications that can be used to find a font. More than one font may match the specifications in a font spec.

`font-spec` **&rest** *arguments* [Function]

Return a new font spec using the specifications in *arguments*, which should come in `property-value` pairs. The possible specifications are as follows:

`:name` The font name (a string), in either XLFD, Fontconfig, or GTK format. See Section "Fonts" in *The GNU Emacs Manual*.

`:family`
`:foundry`
`:weight`
`:slant`
`:width` These have the same meanings as the face attributes of the same name. See Section 11.12.1 [Face Attributes], page 138.

`:size` The font size—either a non-negative integer that specifies the pixel size, or a floating point number that specifies the point size.

`:adstyle` Additional typographic style information for the font, such as '`sans`'. The value should be a string or a symbol.

`:registry`
The charset registry and encoding of the font, such as '`iso8859-1`'. The value should be a string or a symbol.

`:script` The script that the font must support (a symbol).

`:otf` The font must be an OpenType font that supports these OpenType features, provided Emacs is compiled with support for '`libotf`' (a library for performing complex text layout in certain scripts). The value must be a list of the form

```
(script-tag langsys-tag gsub gpos)
```

where *script-tag* is the OpenType script tag symbol; *langsys-tag* is the OpenType language system tag symbol, or `nil` to use the default language system; `gsub` is a list of OpenType GSUB feature tag symbols, or `nil` if none is required; and `gpos` is a list of OpenType GPOS feature tag symbols, or `nil` if none is required. If `gsub` or `gpos` is a list, a `nil` element in that list means that the font must not match any of the remaining tag symbols. The `gpos` element may be omitted.

`font-put` *font-spec property value* [Function]

Set the font property *property* in the font-spec *font-spec* to *value*.

A font entity is a reference to a font that need not be open. Its properties are intermediate between a font object and a font spec: like a font object, and unlike a font spec, it refers to a single, specific font. Unlike a font object, creating a font entity does not load the contents of that font into computer memory.

`find-font` *font-spec* **&optional** *frame* [Function]
: This function returns a font entity that best matches the font spec *font-spec* on frame *frame*. If *frame* is `nil`, it defaults to the selected frame.

`list-fonts` *font-spec* **&optional** *frame num prefer* [Function]
: This function returns a list of all font entities that match the font spec *font-spec*.

  The optional argument *frame*, if non-`nil`, specifies the frame on which the fonts are to be displayed. The optional argument *num*, if non-`nil`, should be an integer that specifies the maximum length of the returned list. The optional argument *prefer*, if non-`nil`, should be another font spec, which is used to control the order of the returned list; the returned font entities are sorted in order of decreasing "closeness" to that font spec.

If you call `set-face-attribute` and pass a font spec, font entity, or font name string as the value of the `:font` attribute, Emacs opens the best "matching" font that is available for display. It then stores the corresponding font object as the actual value of the `:font` attribute for that face.

The following functions can be used to obtain information about a font. For these functions, the *font* argument can be a font object, a font entity, or a font spec.

`font-get` *font property* [Function]
: This function returns the value of the font property *property* for *font*.

  If *font* is a font spec and the font spec does not specify *property*, the return value is `nil`. If *font* is a font object or font entity, the value for the *:script* property may be a list of scripts supported by the font.

`font-face-attributes` *font* **&optional** *frame* [Function]
: This function returns a list of face attributes corresponding to *font*. The optional argument *frame* specifies the frame on which the font is to be displayed. If it is `nil`, the selected frame is used. The return value has the form

```
(:family family :height height :weight weight
   :slant slant :width width)
```

  where the values of *family*, *height*, *weight*, *slant*, and *width* are face attribute values. Some of these key-attribute pairs may be omitted from the list if they are not specified by *font*.

`font-xlfd-name` *font* **&optional** *fold-wildcards* [Function]
: This function returns the XLFD (X Logical Font Descriptor), a string, matching *font*. See Section "Fonts" in *The GNU Emacs Manual*, for information about XLFDs. If the name is too long for an XLFD (which can contain at most 255 characters), the function returns `nil`.

  If the optional argument *fold-wildcards* is non-`nil`, consecutive wildcards in the XLFD are folded into one.

## 11.13 Fringes

On graphical displays, Emacs draws *fringes* next to each window: thin vertical strips down the sides which can display bitmaps indicating truncation, continuation, horizontal scrolling, and so on.

### 11.13.1 Fringe Size and Position

The following buffer-local variables control the position and width of fringes in windows showing that buffer.

`fringes-outside-margins` [Variable]
: The fringes normally appear between the display margins and the window text. If the value is non-`nil`, they appear outside the display margins. See Section 11.15.5 [Display Margins], page 167.

`left-fringe-width` [Variable]
: This variable, if non-`nil`, specifies the width of the left fringe in pixels. A value of `nil` means to use the left fringe width from the window's frame.

`right-fringe-width` [Variable]
: This variable, if non-`nil`, specifies the width of the right fringe in pixels. A value of `nil` means to use the right fringe width from the window's frame.

Any buffer which does not specify values for these variables uses the values specified by the `left-fringe` and `right-fringe` frame parameters (see Section 18.3.3.4 [Layout Parameters], page 349).

The above variables actually take effect via the function `set-window-buffer` (see Section 17.10 [Buffers and Windows], page 309), which calls `set-window-fringes` as a subroutine. If you change one of these variables, the fringe display is not updated in existing windows showing the buffer, unless you call `set-window-buffer` again in each affected window. You can also use `set-window-fringes` to control the fringe display in individual windows.

`set-window-fringes` *window left* **&optional** *right outside-margins* [Function]
: This function sets the fringe widths of window *window*. If *window* is `nil`, the selected window is used.

  The argument *left* specifies the width in pixels of the left fringe, and likewise *right* for the right fringe. A value of `nil` for either one stands for the default width. If *outside-margins* is non-`nil`, that specifies that fringes should appear outside of the display margins.

`window-fringes` **&optional** *window* [Function]
: This function returns information about the fringes of a window *window*. If *window* is omitted or `nil`, the selected window is used. The value has the form `(`*`left-width right-width outside-margins`*`)`.

### 11.13.2 Fringe Indicators

*Fringe indicators* are tiny icons displayed in the window fringe to indicate truncated or continued lines, buffer boundaries, etc.

`indicate-empty-lines` [User Option]
: When this is non-`nil`, Emacs displays a special glyph in the fringe of each empty line at the end of the buffer, on graphical displays. See Section 11.13 [Fringes], page 156. This variable is automatically buffer-local in every buffer.

`indicate-buffer-boundaries` [User Option]

This buffer-local variable controls how the buffer boundaries and window scrolling are indicated in the window fringes.

Emacs can indicate the buffer boundaries—that is, the first and last line in the buffer—with angle icons when they appear on the screen. In addition, Emacs can display an up-arrow in the fringe to show that there is text above the screen, and a down-arrow to show there is text below the screen.

There are three kinds of basic values:

`nil` Don't display any of these fringe icons.

`left` Display the angle icons and arrows in the left fringe.

`right` Display the angle icons and arrows in the right fringe.

any non-alist
Display the angle icons in the left fringe and don't display the arrows.

Otherwise the value should be an alist that specifies which fringe indicators to display and where. Each element of the alist should have the form (*indicator* . *position*). Here, *indicator* is one of `top`, `bottom`, `up`, `down`, and `t` (which covers all the icons not yet specified), while *position* is one of `left`, `right` and `nil`.

For example, `((top . left) (t . right))` places the top angle bitmap in left fringe, and the bottom angle bitmap as well as both arrow bitmaps in right fringe. To show the angle bitmaps in the left fringe, and no arrow bitmaps, use `((top . left) (bottom . left))`.

`fringe-indicator-alist` [Variable]

This buffer-local variable specifies the mapping from logical fringe indicators to the actual bitmaps displayed in the window fringes. The value is an alist of elements (*indicator* . *bitmaps*), where *indicator* specifies a logical indicator type and *bitmaps* specifies the fringe bitmaps to use for that indicator.

Each *indicator* should be one of the following symbols:

`truncation`, `continuation`.
Used for truncation and continuation lines.

`up`, `down`, `top`, `bottom`, `top-bottom`
Used when `indicate-buffer-boundaries` is non-`nil`: `up` and `down` indicate a buffer boundary lying above or below the window edge; `top` and `bottom` indicate the topmost and bottommost buffer text line; and `top-bottom` indicates where there is just one line of text in the buffer.

`empty-line`
Used to indicate empty lines when `indicate-empty-lines` is non-`nil`.

`overlay-arrow`
Used for overlay arrows (see Section 11.13.6 [Overlay Arrow], page 161).

Each *bitmaps* value may be a list of symbols (*left right* [*left1 right1*]). The *left* and *right* symbols specify the bitmaps shown in the left and/or right fringe, for the specific indicator. *left1* and *right1* are specific to the `bottom` and `top-bottom`

indicators, and are used to indicate that the last text line has no final newline. Alternatively, *bitmaps* may be a single symbol which is used in both left and right fringes.

See Section 11.13.4 [Fringe Bitmaps], page 159, for a list of standard bitmap symbols and how to define your own. In addition, `nil` represents the empty bitmap (i.e., an indicator that is not shown).

When `fringe-indicator-alist` has a buffer-local value, and there is no bitmap defined for a logical indicator, or the bitmap is `t`, the corresponding value from the default value of `fringe-indicator-alist` is used.

### 11.13.3 Fringe Cursors

When a line is exactly as wide as the window, Emacs displays the cursor in the right fringe instead of using two lines. Different bitmaps are used to represent the cursor in the fringe depending on the current buffer's cursor type.

`overflow-newline-into-fringe` [User Option]

If this is non-`nil`, lines exactly as wide as the window (not counting the final newline character) are not continued. Instead, when point is at the end of the line, the cursor appears in the right fringe.

`fringe-cursor-alist` [Variable]

This variable specifies the mapping from logical cursor type to the actual fringe bitmaps displayed in the right fringe. The value is an alist where each element has the form `(`*`cursor-type . bitmap`*`)`, which means to use the fringe bitmap *bitmap* to display cursors of type *cursor-type*.

Each *cursor-type* should be one of `box`, `hollow`, `bar`, `hbar`, or `hollow-small`. The first four have the same meanings as in the `cursor-type` frame parameter (see Section 18.3.3.7 [Cursor Parameters], page 351). The `hollow-small` type is used instead of `hollow` when the normal `hollow-rectangle` bitmap is too tall to fit on a specific display line.

Each *bitmap* should be a symbol specifying the fringe bitmap to be displayed for that logical cursor type. See the next subsection for details.

When `fringe-cursor-alist` has a buffer-local value, and there is no bitmap defined for a cursor type, the corresponding value from the default value of `fringes-indicator-alist` is used.

### 11.13.4 Fringe Bitmaps

The *fringe bitmaps* are the actual bitmaps which represent the logical fringe indicators for truncated or continued lines, buffer boundaries, overlay arrows, etc. Each bitmap is represented by a symbol. These symbols are referred to by the variables `fringe-indicator-alist` and `fringe-cursor-alist`, described in the previous subsections.

Lisp programs can also directly display a bitmap in the left or right fringe, by using a `display` property for one of the characters appearing in the line (see Section 11.15.4 [Other Display Specs], page 166). Such a display specification has the form

`(`*`fringe bitmap`* `[`*`face`*`])`

*fringe* is either the symbol `left-fringe` or `right-fringe`. *bitmap* is a symbol identifying the bitmap to display. The optional *face* names a face whose foreground color is used to display the bitmap; this face is automatically merged with the `fringe` face.

Here is a list of the standard fringe bitmaps defined in Emacs, and how they are currently used in Emacs (via `fringe-indicator-alist` and `fringe-cursor-alist`):

`left-arrow`, `right-arrow`
: Used to indicate truncated lines.

`left-curly-arrow`, `right-curly-arrow`
: Used to indicate continued lines.

`right-triangle`, `left-triangle`
: The former is used by overlay arrows. The latter is unused.

`up-arrow`, `down-arrow`, `top-left-angle` `top-right-angle`
`bottom-left-angle`, `bottom-right-angle`
`top-right-angle`, `top-left-angle`
`left-bracket`, `right-bracket`, `top-right-angle`, `top-left-angle`
: Used to indicate buffer boundaries.

`filled-rectangle`, `hollow-rectangle`
`filled-square`, `hollow-square`
`vertical-bar`, `horizontal-bar`
: Used for different types of fringe cursors.

`empty-line`, `exclamation-mark`, `question-mark`, `exclamation-mark`
: Not used by core Emacs features.

The next subsection describes how to define your own fringe bitmaps.

`fringe-bitmaps-at-pos` **&optional** *pos window* [Function]
: This function returns the fringe bitmaps of the display line containing position *pos* in window *window*. The return value has the form `(left right ov)`, where *left* is the symbol for the fringe bitmap in the left fringe (or `nil` if no bitmap), *right* is similar for the right fringe, and *ov* is non-`nil` if there is an overlay arrow in the left fringe.

  The value is `nil` if *pos* is not visible in *window*. If *window* is `nil`, that stands for the selected window. If *pos* is `nil`, that stands for the value of point in *window*.

### 11.13.5 Customizing Fringe Bitmaps

`define-fringe-bitmap` *bitmap bits* **&optional** *height width align* [Function]
: This function defines the symbol *bitmap* as a new fringe bitmap, or replaces an existing bitmap with that name.

  The argument *bits* specifies the image to use. It should be either a string or a vector of integers, where each element (an integer) corresponds to one row of the bitmap. Each bit of an integer corresponds to one pixel of the bitmap, where the low bit corresponds to the rightmost pixel of the bitmap.

  The height is normally the length of *bits*. However, you can specify a different height with non-`nil` *height*. The width is normally 8, but you can specify a different width with non-`nil` *width*. The width must be an integer between 1 and 16.

The argument *align* specifies the positioning of the bitmap relative to the range of rows where it is used; the default is to center the bitmap. The allowed values are `top`, `center`, or `bottom`.

The *align* argument may also be a list `(align periodic)` where *align* is interpreted as described above. If *periodic* is non-`nil`, it specifies that the rows in `bits` should be repeated enough times to reach the specified height.

`destroy-fringe-bitmap` *bitmap* [Function]
This function destroy the fringe bitmap identified by *bitmap*. If *bitmap* identifies a standard fringe bitmap, it actually restores the standard definition of that bitmap, instead of eliminating it entirely.

`set-fringe-bitmap-face` *bitmap* **&optional** *face* [Function]
This sets the face for the fringe bitmap *bitmap* to *face*. If *face* is `nil`, it selects the `fringe` face. The bitmap's face controls the color to draw it in.

*face* is merged with the `fringe` face, so normally *face* should specify only the foreground color.

### 11.13.6 The Overlay Arrow

The *overlay arrow* is useful for directing the user's attention to a particular line in a buffer. For example, in the modes used for interface to debuggers, the overlay arrow indicates the line of code about to be executed. This feature has nothing to do with *overlays* (see Section 11.9 [Overlays], page 128).

`overlay-arrow-string` [Variable]
This variable holds the string to display to call attention to a particular line, or `nil` if the arrow feature is not in use. On a graphical display the contents of the string are ignored; instead a glyph is displayed in the fringe area to the left of the display area.

`overlay-arrow-position` [Variable]
This variable holds a marker that indicates where to display the overlay arrow. It should point at the beginning of a line. On a non-graphical display the arrow text appears at the beginning of that line, overlaying any text that would otherwise appear. Since the arrow is usually short, and the line usually begins with indentation, normally nothing significant is overwritten.

The overlay-arrow string is displayed in any given buffer if the value of `overlay-arrow-position` in that buffer points into that buffer. Thus, it is possible to display multiple overlay arrow strings by creating buffer-local bindings of `overlay-arrow-position`. However, it is usually cleaner to use `overlay-arrow-variable-list` to achieve this result.

You can do a similar job by creating an overlay with a `before-string` property. See Section 11.9.2 [Overlay Properties], page 131.

You can define multiple overlay arrows via the variable `overlay-arrow-variable-list`.

`overlay-arrow-variable-list` [Variable]
: This variable's value is a list of variables, each of which specifies the position of an overlay arrow. The variable `overlay-arrow-position` has its normal meaning because it is on this list.

Each variable on this list can have properties `overlay-arrow-string` and `overlay-arrow-bitmap` that specify an overlay arrow string (for text terminals) or fringe bitmap (for graphical terminals) to display at the corresponding overlay arrow position. If either property is not set, the default `overlay-arrow-string` or `overlay-arrow` fringe indicator is used.

## 11.14 Scroll Bars

Normally the frame parameter `vertical-scroll-bars` controls whether the windows in the frame have vertical scroll bars, and whether they are on the left or right. The frame parameter `scroll-bar-width` specifies how wide they are (`nil` meaning the default). See Section 18.3.3.4 [Layout Parameters], page 349.

`frame-current-scroll-bars` **&optional** *frame* [Function]
: This function reports the scroll bar type settings for frame *frame*. The value is a cons cell (*vertical-type* . *horizontal-type*), where *vertical-type* is either `left`, `right`, or `nil` (which means no scroll bar.) *horizontal-type* is meant to specify the horizontal scroll bar type, but since they are not implemented, it is always `nil`.

You can enable or disable scroll bars for a particular buffer, by setting the variable `vertical-scroll-bar`. This variable automatically becomes buffer-local when set. The possible values are `left`, `right`, `t`, which means to use the frame's default, and `nil` for no scroll bar.

You can also control this for individual windows. Call the function `set-window-scroll-bars` to specify what to do for a specific window:

`set-window-scroll-bars` *window width* **&optional** *vertical-type horizontal-type* [Function]
: This function sets the width and type of scroll bars for window *window*.

  *width* specifies the scroll bar width in pixels (`nil` means use the width specified for the frame). *vertical-type* specifies whether to have a vertical scroll bar and, if so, where. The possible values are `left`, `right` and `nil`, just like the values of the `vertical-scroll-bars` frame parameter.

  The argument *horizontal-type* is meant to specify whether and where to have horizontal scroll bars, but since they are not implemented, it has no effect. If *window* is `nil`, the selected window is used.

`window-scroll-bars` **&optional** *window* [Function]
: Report the width and type of scroll bars specified for *window*. If *window* is omitted or `nil`, the selected window is used. The value is a list of the form (*width cols vertical-type horizontal-type*). The value *width* is the value that was specified for the width (which may be `nil`); *cols* is the number of columns that the scroll bar actually occupies.

  *horizontal-type* is not actually meaningful.

If you don't specify these values for a window with `set-window-scroll-bars`, the buffer-local variables `scroll-bar-mode` and `scroll-bar-width` in the buffer being displayed control the window's vertical scroll bars. The function `set-window-buffer` examines these variables. If you change them in a buffer that is already visible in a window, you can make the window take note of the new values by calling `set-window-buffer` specifying the same buffer that is already displayed.

`scroll-bar-mode` [User Option]
: This variable, always local in all buffers, controls whether and where to put scroll bars in windows displaying the buffer. The possible values are `nil` for no scroll bar, `left` to put a scroll bar on the left, and `right` to put a scroll bar on the right.

`window-current-scroll-bars` **&optional** *window* [Function]
: This function reports the scroll bar type for window *window*. If *window* is omitted or `nil`, the selected window is used. The value is a cons cell `(`*`vertical-type`*` . `*`horizontal-type`*`)`. Unlike `window-scroll-bars`, this reports the scroll bar type actually used, once frame defaults and `scroll-bar-mode` are taken into account.

`scroll-bar-width` [Variable]
: This variable, always local in all buffers, specifies the width of the buffer's scroll bars, measured in pixels. A value of `nil` means to use the value specified by the frame.

## 11.15 The `display` Property

The `display` text property (or overlay property) is used to insert images into text, and to control other aspects of how text displays. The value of the `display` property should be a display specification, or a list or vector containing several display specifications. Display specifications in the same `display` property value generally apply in parallel to the text they cover.

If several sources (overlays and/or a text property) specify values for the `display` property, only one of the values takes effect, following the rules of `get-char-property`. See Section 22.19.1 [Examining Properties], page 489.

The rest of this section describes several kinds of display specifications and what they mean.

### 11.15.1 Display Specs That Replace The Text

Some kinds of display specifications specify something to display instead of the text that has the property. These are called *replacing* display specifications. Emacs does not allow the user to interactively move point into the middle of buffer text that is replaced in this way.

If a list of display specifications includes more than one replacing display specification, the first overrides the rest. Replacing display specifications make most other display specifications irrelevant, since those don't apply to the replacement.

For replacing display specifications, "the text that has the property" means all the consecutive characters that have the same Lisp object as their `display` property; these characters are replaced as a single unit. If two characters have different Lisp objects as their `display` properties (i.e., objects which are not `eq`), they are handled separately.

Here is an example which illustrates this point. A string serves as a replacing display specification, which replaces the text that has the property with the specified string (see Section 11.15.4 [Other Display Specs], page 166). Consider the following function:

```
(defun foo ()
  (dotimes (i 5)
    (let ((string (concat "A"))
          (start (+ i i (point-min))))
      (put-text-property start (1+ start) 'display string)
      (put-text-property start (+ 2 start) 'display string))))
```

This function gives each of the first ten characters in the buffer a `display` property which is a string `"A"`, but they don't all get the same string object. The first two characters get the same string object, so they are replaced with one '`A`'; the fact that the display property was assigned in two separate calls to `put-text-property` is irrelevant. Similarly, the next two characters get a second string (`concat` creates a new string object), so they are replaced with one '`A`'; and so on. Thus, the ten characters appear as five A's.

### 11.15.2 Specified Spaces

To display a space of specified width and/or height, use a display specification of the form `(space . `*`props`*`)`, where *props* is a property list (a list of alternating properties and values). You can put this property on one or more consecutive characters; a space of the specified height and width is displayed in place of *all* of those characters. These are the properties you can use in *props* to specify the weight of the space:

`:width` *width*
: If *width* is an integer or floating point number, it specifies that the space width should be *width* times the normal character width. *width* can also be a *pixel width* specification (see Section 11.15.3 [Pixel Specification], page 165).

`:relative-width` *factor*
: Specifies that the width of the stretch should be computed from the first character in the group of consecutive characters that have the same `display` property. The space width is the width of that character, multiplied by *factor*.

`:align-to` *hpos*
: Specifies that the space should be wide enough to reach *hpos*. If *hpos* is a number, it is measured in units of the normal character width. *hpos* can also be a *pixel width* specification (see Section 11.15.3 [Pixel Specification], page 165).

You should use one and only one of the above properties. You can also specify the height of the space, with these properties:

`:height` *height*
: Specifies the height of the space. If *height* is an integer or floating point number, it specifies that the space height should be *height* times the normal character height. The *height* may also be a *pixel height* specification (see Section 11.15.3 [Pixel Specification], page 165).

`:relative-height` *factor*
: Specifies the height of the space, multiplying the ordinary height of the text having this display specification by *factor*.

`:ascent` *ascent*
: If the value of *ascent* is a non-negative number no greater than 100, it specifies that *ascent* percent of the height of the space should be considered as the ascent of the space—that is, the part above the baseline. The ascent may also be specified in pixel units with a *pixel ascent* specification (see Section 11.15.3 [Pixel Specification], page 165).

Don't use both `:height` and `:relative-height` together.

The `:width` and `:align-to` properties are supported on non-graphic terminals, but the other space properties in this section are not.

Note that space properties are treated as paragraph separators for the purposes of reordering bidirectional text for display. See Section 11.23 [Bidirectional Display], page 195, for the details.

### 11.15.3 Pixel Specification for Spaces

The value of the `:width`, `:align-to`, `:height`, and `:ascent` properties can be a special kind of expression that is evaluated during redisplay. The result of the evaluation is used as an absolute number of pixels.

The following expressions are supported:

```
expr ::= num | (num) | unit | elem | pos | image | form
num  ::= integer | float | symbol
unit ::= in | mm | cm | width | height
elem ::= left-fringe | right-fringe | left-margin | right-margin
       | scroll-bar | text
pos  ::= left | center | right
form ::= (num . expr) | (op expr ...)
op   ::= + | -
```

The form *num* specifies a fraction of the default frame font height or width. The form `(`*num*`)` specifies an absolute number of pixels. If *num* is a symbol, *symbol*, its buffer-local variable binding is used.

The `in`, `mm`, and `cm` units specify the number of pixels per inch, millimeter, and centimeter, respectively. The `width` and `height` units correspond to the default width and height of the current face. An image specification `image` corresponds to the width or height of the image.

The elements `left-fringe`, `right-fringe`, `left-margin`, `right-margin`, `scroll-bar`, and `text` specify to the width of the corresponding area of the window.

The `left`, `center`, and `right` positions can be used with `:align-to` to specify a position relative to the left edge, center, or right edge of the text area.

Any of the above window elements (except `text`) can also be used with `:align-to` to specify that the position is relative to the left edge of the given area. Once the base offset for a relative position has been set (by the first occurrence of one of these symbols), further occurrences of these symbols are interpreted as the width of the specified area. For example, to align to the center of the left-margin, use

```
:align-to (+ left-margin (0.5 . left-margin))
```

If no specific base offset is set for alignment, it is always relative to the left edge of the text area. For example, '`:align-to 0`' in a header-line aligns with the first text column in the text area.

A value of the form `(num . expr)` stands for the product of the values of *num* and *expr*. For example, `(2 . in)` specifies a width of 2 inches, while `(0.5 . image)` specifies half the width (or height) of the specified image.

The form `(+ expr ...)` adds up the value of the expressions. The form `(- expr ...)` negates or subtracts the value of the expressions.

### 11.15.4 Other Display Specifications

Here are the other sorts of display specifications that you can use in the `display` text property.

*string* Display *string* instead of the text that has this property.

Recursive display specifications are not supported—*string*'s `display` properties, if any, are not used.

`(image . image-props)`
This kind of display specification is an image descriptor (see Section 11.16 [Images], page 168). When used as a display specification, it means to display the image instead of the text that has the display specification.

`(slice x y width height)`
This specification together with `image` specifies a *slice* (a partial area) of the image to display. The elements *y* and *x* specify the top left corner of the slice, within the image; *width* and *height* specify the width and height of the slice. Integer values are numbers of pixels. A floating point number in the range 0.0–1.0 stands for that fraction of the width or height of the entire image.

`((margin nil) string)`
A display specification of this form means to display *string* instead of the text that has the display specification, at the same position as that text. It is equivalent to using just *string*, but it is done as a special case of marginal display (see Section 11.15.5 [Display Margins], page 167).

`(left-fringe bitmap [face])`
`(right-fringe bitmap [face])`
This display specification on any character of a line of text causes the specified *bitmap* be displayed in the left or right fringes for that line, instead of the characters that have the display specification. The optional *face* specifies the colors to be used for the bitmap. See Section 11.13.4 [Fringe Bitmaps], page 159, for the details.

`(space-width factor)`
This display specification affects all the space characters within the text that has the specification. It displays all of these spaces *factor* times as wide as normal. The element *factor* should be an integer or float. Characters other than spaces are not affected at all; in particular, this has no effect on tab characters.

`(height height)`
This display specification makes the text taller or shorter. Here are the possibilities for *height*:

`(+ n)` This means to use a font that is *n* steps larger. A "step" is defined by the set of available fonts—specifically, those that match what was otherwise specified for this text, in all attributes except height. Each size for which a suitable font is available counts as another step. *n* should be an integer.

`(- n)` This means to use a font that is *n* steps smaller.

a number, *factor*
: A number, *factor*, means to use a font that is *factor* times as tall as the default font.

a symbol, *function*
: A symbol is a function to compute the height. It is called with the current height as argument, and should return the new height to use.

anything else, *form*
: If the *height* value doesn't fit the previous possibilities, it is a form. Emacs evaluates it to get the new height, with the symbol `height` bound to the current specified font height.

`(raise` *factor*`)`
: This kind of display specification raises or lowers the text it applies to, relative to the baseline of the line.

  *factor* must be a number, which is interpreted as a multiple of the height of the affected text. If it is positive, that means to display the characters raised. If it is negative, that means to display them lower down.

  If the text also has a `height` display specification, that does not affect the amount of raising or lowering, which is based on the faces used for the text.

You can make any display specification conditional. To do that, package it in another list of the form `(when` *condition* `.` *spec*`)`. Then the specification *spec* applies only when *condition* evaluates to a non-`nil` value. During the evaluation, `object` is bound to the string or buffer having the conditional `display` property. `position` and `buffer-position` are bound to the position within `object` and the buffer position where the `display` property was found, respectively. Both positions can be different when `object` is a string.

### 11.15.5 Displaying in the Margins

A buffer can have blank areas called *display margins* on the left and on the right. Ordinary text never appears in these areas, but you can put things into the display margins using the `display` property. There is currently no way to make text or images in the margin mouse-sensitive.

The way to display something in the margins is to specify it in a margin display specification in the `display` property of some text. This is a replacing display specification, meaning that the text you put it on does not get displayed; the margin display appears, but that text does not.

A margin display specification looks like `((margin right-margin)` *spec*`)` or `((margin left-margin)` *spec*`)`. Here, *spec* is another display specification that says what to display in the margin. Typically it is a string of text to display, or an image descriptor.

To display something in the margin *in association with* certain buffer text, without altering or preventing the display of that text, put a `before-string` property on the text and put the margin display specification on the contents of the before-string.

Before the display margins can display anything, you must give them a nonzero width. The usual way to do that is to set these variables:

`left-margin-width` [Variable]
: This variable specifies the width of the left margin. It is buffer-local in all buffers.

`right-margin-width` [Variable]
: This variable specifies the width of the right margin. It is buffer-local in all buffers.

Setting these variables does not immediately affect the window. These variables are checked when a new buffer is displayed in the window. Thus, you can make changes take effect by calling `set-window-buffer`.

You can also set the margin widths immediately.

`set-window-margins` *window left* **&optional** *right* [Function]
: This function specifies the margin widths for window *window*. The argument *left* controls the left margin and *right* controls the right margin (default `0`).

`window-margins` **&optional** *window* [Function]
: This function returns the left and right margins of *window* as a cons cell of the form `(`*left* `.` *right*`)`. If *window* is `nil`, the selected window is used.

## 11.16 Images

To display an image in an Emacs buffer, you must first create an image descriptor, then use it as a display specifier in the `display` property of text that is displayed (see Section 11.15 [Display Property], page 163).

Emacs is usually able to display images when it is run on a graphical terminal. Images cannot be displayed in a text terminal, on certain graphical terminals that lack the support for this, or if Emacs is compiled without image support. You can use the function `display-images-p` to determine if images can in principle be displayed (see Section 18.23 [Display Feature Testing], page 370).

### 11.16.1 Image Formats

Emacs can display a number of different image formats. Some of these image formats are supported only if particular support libraries are installed. On some platforms, Emacs can load support libraries on demand; if so, the variable `dynamic-library-alist` can be used to modify the set of known names for these dynamic libraries. See ⟨undefined⟩ [Dynamic Libraries], page ⟨undefined⟩.

Supported image formats (and the required support libraries) include PBM and XBM (which do not depend on support libraries and are always available), XPM (`libXpm`), GIF (`libgif` or `libungif`), PostScript (`gs`), JPEG (`libjpeg`), TIFF (`libtiff`), PNG (`libpng`), and SVG (`librsvg`).

Each of these image formats is associated with an *image type symbol*. The symbols for the above formats are, respectively, `pbm`, `xbm`, `xpm`, `gif`, `postscript`, `jpeg`, `tiff`, `png`, and `svg`.

Furthermore, if you build Emacs with ImageMagick (`libMagickWand`) support, Emacs can display any image format that ImageMagick can. See Section 11.16.8 [ImageMagick Images], page 173. All images displayed via ImageMagick have type symbol `imagemagick`.

`image-types` [Variable]
: This variable contains a list of type symbols for image formats which are potentially supported in the current configuration.

  "Potentially" means that Emacs knows about the image types, not necessarily that they can be used (for example, they could depend on unavailable dynamic libraries). To know which image types are really available, use `image-type-available-p`.

`image-type-available-p` *type* [Function]
: This function returns non-`nil` if images of type *type* can be loaded and displayed. *type* must be an image type symbol.

  For image types whose support libraries are statically linked, this function always returns `t`. For image types whose support libraries are dynamically loaded, it returns `t` if the library could be loaded and `nil` otherwise.

### 11.16.2 Image Descriptors

An *image descriptor* is a list which specifies the underlying data for an image, and how to display it. It is typically used as the value of a `display` overlay or text property (see Section 11.15.4 [Other Display Specs], page 166); but See Section 11.16.11 [Showing Images], page 176, for convenient helper functions to insert images into buffers.

Each image descriptor has the form `(image . props)`, where *props* is a property list of alternating keyword symbols and values, including at least the pair `:type TYPE` which specifies the image type.

The following is a list of properties that are meaningful for all image types (there are also properties which are meaningful only for certain image types, as documented in the following subsections):

`:type` *type*
: The image type. Every image descriptor must include this property.

`:file` *file*
: This says to load the image from file *file*. If *file* is not an absolute file name, it is expanded in `data-directory`.

`:data` *data*
: This specifies the raw image data. Each image descriptor must have either `:data` or `:file`, but not both.

  For most image types, the value of a `:data` property should be a string containing the image data. Some image types do not support `:data`; for some others, `:data` alone is not enough, so you need to use other image properties along with `:data`. See the following subsections for details.

`:margin` *margin*
: This specifies how many pixels to add as an extra margin around the image. The value, *margin*, must be a non-negative number, or a pair `(x . y)` of such

numbers. If it is a pair, *x* specifies how many pixels to add horizontally, and *y* specifies how many pixels to add vertically. If `:margin` is not specified, the default is zero.

`:ascent` *ascent*
: This specifies the amount of the image's height to use for its ascent—that is, the part above the baseline. The value, *ascent*, must be a number in the range 0 to 100, or the symbol `center`.

  If *ascent* is a number, that percentage of the image's height is used for its ascent.

  If *ascent* is `center`, the image is vertically centered around a centerline which would be the vertical centerline of text drawn at the position of the image, in the manner specified by the text properties and overlays that apply to the image.

  If this property is omitted, it defaults to 50.

`:relief` *relief*
: This adds a shadow rectangle around the image. The value, *relief*, specifies the width of the shadow lines, in pixels. If *relief* is negative, shadows are drawn so that the image appears as a pressed button; otherwise, it appears as an unpressed button.

`:conversion` *algorithm*
: This specifies a conversion algorithm that should be applied to the image before it is displayed; the value, *algorithm*, specifies which algorithm.

  `laplace`
  `emboss`
  : Specifies the Laplace edge detection algorithm, which blurs out small differences in color while highlighting larger differences. People sometimes consider this useful for displaying the image for a "disabled" button.

  `(edge-detection :matrix` *matrix* `:color-adjust` *adjust*`)`
  : Specifies a general edge-detection algorithm. *matrix* must be either a nine-element list or a nine-element vector of numbers. A pixel at position $x/y$ in the transformed image is computed from original pixels around that position. *matrix* specifies, for each pixel in the neighborhood of $x/y$, a factor with which that pixel will influence the transformed pixel; element 0 specifies the factor for the pixel at $x-1/y-1$, element 1 the factor for the pixel at $x/y-1$ etc., as shown below:

    $$\begin{pmatrix} x-1/y-1 & x/y-1 & x+1/y-1 \\ x-1/y & x/y & x+1/y \\ x-1/y+1 & x/y+1 & x+1/y+1 \end{pmatrix}$$

    The resulting pixel is computed from the color intensity of the color resulting from summing up the RGB values of surrounding pixels, multiplied by the specified factors, and dividing that sum by the sum of the factors' absolute values.

Laplace edge-detection currently uses a matrix of

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

Emboss edge-detection uses a matrix of

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & -2 \end{pmatrix}$$

`disabled` Specifies transforming the image so that it looks "disabled".

`:mask` *mask*

If *mask* is `heuristic` or `(heuristic` *bg*`)`, build a clipping mask for the image, so that the background of a frame is visible behind the image. If *bg* is not specified, or if *bg* is `t`, determine the background color of the image by looking at the four corners of the image, assuming the most frequently occurring color from the corners is the background color of the image. Otherwise, *bg* must be a list `(`*red green blue*`)` specifying the color to assume for the background of the image.

If *mask* is `nil`, remove a mask from the image, if it has one. Images in some formats include a mask which can be removed by specifying `:mask nil`.

`:pointer` *shape*

This specifies the pointer shape when the mouse pointer is over this image. See Section 18.17 [Pointer Shape], page 365, for available pointer shapes.

`:map` *map* This associates an image map of *hot spots* with this image.

An image map is an alist where each element has the format `(`*area id plist*`)`. An *area* is specified as either a rectangle, a circle, or a polygon.

A rectangle is a cons `(rect . ((`*x0 . y0*`) . (`*x1 . y1*`)))` which specifies the pixel coordinates of the upper left and bottom right corners of the rectangle area.

A circle is a cons `(circle . ((`*x0 . y0*`) . `*r*`))` which specifies the center and the radius of the circle; *r* may be a float or integer.

A polygon is a cons `(poly . [`*x0 y0 x1 y1 ...*`])` where each pair in the vector describes one corner in the polygon.

When the mouse pointer lies on a hot-spot area of an image, the *plist* of that hot-spot is consulted; if it contains a `help-echo` property, that defines a tool-tip for the hot-spot, and if it contains a `pointer` property, that defines the shape of the mouse cursor when it is on the hot-spot. See Section 18.17 [Pointer Shape], page 365, for available pointer shapes.

When you click the mouse when the mouse pointer is over a hot-spot, an event is composed by combining the *id* of the hot-spot with the mouse event; for instance, `[area4 mouse-1]` if the hot-spot's *id* is `area4`.

`image-mask-p` *spec* **&optional** *frame* [Function]
: This function returns `t` if image *spec* has a mask bitmap. *frame* is the frame on which the image will be displayed. *frame* `nil` or omitted means to use the selected frame (see Section 18.9 [Input Focus], page 358).

### 11.16.3 XBM Images

To use XBM format, specify `xbm` as the image type. This image format doesn't require an external library, so images of this type are always supported.

Additional image properties supported for the `xbm` image type are:

`:foreground` *foreground*
: The value, *foreground*, should be a string specifying the image foreground color, or `nil` for the default color. This color is used for each pixel in the XBM that is 1. The default is the frame's foreground color.

`:background` *background*
: The value, *background*, should be a string specifying the image background color, or `nil` for the default color. This color is used for each pixel in the XBM that is 0. The default is the frame's background color.

If you specify an XBM image using data within Emacs instead of an external file, use the following three properties:

`:data` *data*
: The value, *data*, specifies the contents of the image. There are three formats you can use for *data*:

  - A vector of strings or bool-vectors, each specifying one line of the image. Do specify `:height` and `:width`.
  - A string containing the same byte sequence as an XBM file would contain. You must not specify `:height` and `:width` in this case, because omitting them is what indicates the data has the format of an XBM file. The file contents specify the height and width of the image.
  - A string or a bool-vector containing the bits of the image (plus perhaps some extra bits at the end that will not be used). It should contain at least *width* * `height` bits. In this case, you must specify `:height` and `:width`, both to indicate that the string contains just the bits rather than a whole XBM file, and to specify the size of the image.

`:width` *width*
: The value, *width*, specifies the width of the image, in pixels.

`:height` *height*
: The value, *height*, specifies the height of the image, in pixels.

### 11.16.4 XPM Images

To use XPM format, specify `xpm` as the image type. The additional image property `:color-symbols` is also meaningful with the `xpm` image type:

`:color-symbols` *symbols*
: The value, *symbols*, should be an alist whose elements have the form `(`*name* `.` *color*`)`. In each element, *name* is the name of a color as it appears in the image file, and *color* specifies the actual color to use for displaying that name.

### 11.16.5 GIF Images

For GIF images, specify image type `gif`.

`:index` *index*
: You can use `:index` to specify image number *index* from a GIF file that contains more than one image. If the GIF file doesn't contain an image with the specified index, the image displays as a hollow box. GIF files with more than one image can be animated, see Section 11.16.12 [Animated Images], page 178.

### 11.16.6 TIFF Images

For TIFF images, specify image type `tiff`.

`:index` *index*
: You can use `:index` to specify image number *index* from a TIFF file that contains more than one image. If the TIFF file doesn't contain an image with the specified index, the image displays as a hollow box.

### 11.16.7 PostScript Images

To use PostScript for an image, specify image type `postscript`. This works only if you have Ghostscript installed. You must always use these three properties:

`:pt-width` *width*
: The value, *width*, specifies the width of the image measured in points (1/72 inch). *width* must be an integer.

`:pt-height` *height*
: The value, *height*, specifies the height of the image in points (1/72 inch). *height* must be an integer.

`:bounding-box` *box*
: The value, *box*, must be a list or vector of four integers, which specifying the bounding box of the PostScript image, analogous to the '`BoundingBox`' comment found in PostScript files.

```
%%BoundingBox: 22 171 567 738
```

### 11.16.8 ImageMagick Images

If you build Emacs with ImageMagick support, you can use the ImageMagick library to load many image formats (see Section "File Conveniences" in *The GNU Emacs Manual*). The image type symbol for images loaded via ImageMagick is `imagemagick`, regardless of the actual underlying image format.

`imagemagick-types` [Function]
: This function returns a list of image file extensions supported by the current ImageMagick installation. Each list element is a symbol representing an internal ImageMagick name for an image type, such as `BMP` for '`.bmp`' images.

`imagemagick-enabled-types` [User Option]
: The value of this variable is a list of ImageMagick image types which Emacs may attempt to render using ImageMagick. Each list element should be one of the symbols in the list returned by `imagemagick-types`, or an equivalent string. Alternatively, a value of `t` enables ImageMagick for all possible image types. Regardless of the value of this variable, `imagemagick-types-inhibit` (see below) takes precedence.

`imagemagick-types-inhibit` [User Option]
: The value of this variable lists the ImageMagick image types which should never be rendered using ImageMagick, regardless of the value of `imagemagick-enabled-types`. A value of `t` disables ImageMagick entirely.

Images loaded with ImageMagick support the following additional image descriptor properties:

`:background` *background*
: *background*, if non-`nil`, should be a string specifying a color, which is used as the image's background color if the image supports transparency. If the value is `nil`, it defaults to the frame's background color.

`:width, :height`
: The `:width` and `:height` keywords are used for scaling the image. If only one of them is specified, the other one will be calculated so as to preserve the aspect ratio. If both are specified, aspect ratio may not be preserved.

`:rotation`
: Specifies a rotation angle in degrees.

`:index` This has the same meaning as it does for GIF images (see Section 11.16.5 [GIF Images], page 173), i.e., it specifies which image to view inside an image bundle file format such as DJVM. You can use the `image-metadata` function to retrieve the total number of images in an image bundle.

### 11.16.9 Other Image Types

For PBM images, specify image type `pbm`. Color, gray-scale and monochromatic images are supported. For mono PBM images, two additional image properties are supported.

`:foreground` *foreground*
: The value, *foreground*, should be a string specifying the image foreground color, or `nil` for the default color. This color is used for each pixel in the PBM that is 1. The default is the frame's foreground color.

`:background` *background*
: The value, *background*, should be a string specifying the image background color, or `nil` for the default color. This color is used for each pixel in the PBM that is 0. The default is the frame's background color.

For JPEG images, specify image type `jpeg`.

For TIFF images, specify image type `tiff`.

For PNG images, specify image type `png`.

For SVG images, specify image type `svg`.

### 11.16.10 Defining Images

The functions `create-image`, `defimage` and `find-image` provide convenient ways to create image descriptors.

`create-image` *file-or-data* **&optional** *type data-p* **&rest** *props* [Function]

This function creates and returns an image descriptor which uses the data in *file-or-data*. *file-or-data* can be a file name or a string containing the image data; *data-p* should be `nil` for the former case, non-`nil` for the latter case.

The optional argument *type* is a symbol specifying the image type. If *type* is omitted or `nil`, `create-image` tries to determine the image type from the file's first few bytes, or else from the file's name.

The remaining arguments, *props*, specify additional image properties—for example,

```
(create-image "foo.xpm" 'xpm nil :heuristic-mask t)
```

The function returns `nil` if images of this type are not supported. Otherwise it returns an image descriptor.

`defimage` *symbol specs* **&optional** *doc* [Macro]

This macro defines *symbol* as an image name. The arguments *specs* is a list which specifies how to display the image. The third argument, *doc*, is an optional documentation string.

Each argument in *specs* has the form of a property list, and each one should specify at least the `:type` property and either the `:file` or the `:data` property. The value of `:type` should be a symbol specifying the image type, the value of `:file` is the file to load the image from, and the value of `:data` is a string containing the actual image data. Here is an example:

```
(defimage test-image
  ((:type xpm :file "~/test1.xpm")
   (:type xbm :file "~/test1.xbm")))
```

`defimage` tests each argument, one by one, to see if it is usable—that is, if the type is supported and the file exists. The first usable argument is used to make an image descriptor which is stored in *symbol*.

If none of the alternatives will work, then *symbol* is defined as `nil`.

`find-image` *specs* [Function]

This function provides a convenient way to find an image satisfying one of a list of image specifications *specs*.

Each specification in *specs* is a property list with contents depending on image type. All specifications must at least contain the properties `:type` *type* and either `:file` *file* or `:data` *DATA*, where *type* is a symbol specifying the image type, e.g., `xbm`, *file* is the file to load the image from, and *data* is a string containing the actual image data. The first specification in the list whose *type* is supported, and *file* exists, is used to construct the image specification to be returned. If no specification is satisfied, `nil` is returned.

The image is looked for in `image-load-path`.

`image-load-path` [Variable]

This variable's value is a list of locations in which to search for image files. If an element is a string or a variable symbol whose value is a string, the string is taken to be the name of a directory to search. If an element is a variable symbol whose value is a list, that is taken to be a list of directory names to search.

The default is to search in the '`images`' subdirectory of the directory specified by `data-directory`, then the directory specified by `data-directory`, and finally in the directories in `load-path`. Subdirectories are not automatically included in the search, so if you put an image file in a subdirectory, you have to supply the subdirectory name explicitly. For example, to find the image '`images/foo/bar.xpm`' within `data-directory`, you should specify the image as follows:

```
(defimage foo-image '((:type xpm :file "foo/bar.xpm")))
```

`image-load-path-for-library` *library image* **&optional** *path no-error* [Function]

This function returns a suitable search path for images used by the Lisp package *library*.

The function searches for *image* first using `image-load-path`, excluding '`data-directory/images`', and then in `load-path`, followed by a path suitable for *library*, which includes '`../../etc/images`' and '`../etc/images`' relative to the library file itself, and finally in '`data-directory/images`'.

Then this function returns a list of directories which contains first the directory in which *image* was found, followed by the value of `load-path`. If *path* is given, it is used instead of `load-path`.

If *no-error* is non-`nil` and a suitable path can't be found, don't signal an error. Instead, return a list of directories as before, except that `nil` appears in place of the image directory.

Here is an example of using `image-load-path-for-library`:

```
(defvar image-load-path) ; shush compiler
(let* ((load-path (image-load-path-for-library
                   "mh-e" "mh-logo.xpm"))
       (image-load-path (cons (car load-path)
                              image-load-path)))
  (mh-tool-bar-folder-buttons-init))
```

### 11.16.11 Showing Images

You can use an image descriptor by setting up the `display` property yourself, but it is easier to use the functions in this section.

`insert-image` *image* **&optional** *string area slice* [Function]

This function inserts *image* in the current buffer at point. The value *image* should be an image descriptor; it could be a value returned by `create-image`, or the value of a symbol defined with `defimage`. The argument *string* specifies the text to put in the buffer to hold the image. If it is omitted or `nil`, `insert-image` uses `" "` by default.

The argument *area* specifies whether to put the image in a margin. If it is `left-margin`, the image appears in the left margin; `right-margin` specifies the right mar-

gin. If *area* is `nil` or omitted, the image is displayed at point within the buffer's text.

The argument *slice* specifies a slice of the image to insert. If *slice* is `nil` or omitted the whole image is inserted. Otherwise, *slice* is a list `(x y width height)` which specifies the *x* and *y* positions and *width* and *height* of the image area to insert. Integer values are in units of pixels. A floating point number in the range 0.0–1.0 stands for that fraction of the width or height of the entire image.

Internally, this function inserts *string* in the buffer, and gives it a `display` property which specifies *image*. See Section 11.15 [Display Property], page 163.

`insert-sliced-image` *image* **&optional** *string area rows cols* [Function]

This function inserts *image* in the current buffer at point, like `insert-image`, but splits the image into *rows*x*cols* equally sized slices.

If an image is inserted "sliced", Emacs displays each slice as a separate image, and allow more intuitive scrolling up/down, instead of jumping up/down the entire image when paging through a buffer that displays (large) images.

`put-image` *image pos* **&optional** *string area* [Function]

This function puts image *image* in front of *pos* in the current buffer. The argument *pos* should be an integer or a marker. It specifies the buffer position where the image should appear. The argument *string* specifies the text that should hold the image as an alternative to the default.

The argument *image* must be an image descriptor, perhaps returned by `create-image` or stored by `defimage`.

The argument *area* specifies whether to put the image in a margin. If it is `left-margin`, the image appears in the left margin; `right-margin` specifies the right margin. If *area* is `nil` or omitted, the image is displayed at point within the buffer's text.

Internally, this function creates an overlay, and gives it a `before-string` property containing text that has a `display` property whose value is the image. (Whew!)

`remove-images` *start end* **&optional** *buffer* [Function]

This function removes images in *buffer* between positions *start* and *end*. If *buffer* is omitted or `nil`, images are removed from the current buffer.

This removes only images that were put into *buffer* the way `put-image` does it, not images that were inserted with `insert-image` or in other ways.

`image-size` *spec* **&optional** *pixels frame* [Function]

This function returns the size of an image as a pair `(width . height)`. *spec* is an image specification. *pixels* non-`nil` means return sizes measured in pixels, otherwise return sizes measured in canonical character units (fractions of the width/height of the frame's default font). *frame* is the frame on which the image will be displayed. *frame* null or omitted means use the selected frame (see Section 18.9 [Input Focus], page 358).

`max-image-size` [Variable]

This variable is used to define the maximum size of image that Emacs will load. Emacs will refuse to load (and display) any image that is larger than this limit.

If the value is an integer, it directly specifies the maximum image height and width, measured in pixels. If it is a floating point number, it specifies the maximum image height and width as a ratio to the frame height and width. If the value is non-numeric, there is no explicit limit on the size of images.

The purpose of this variable is to prevent unreasonably large images from accidentally being loaded into Emacs. It only takes effect the first time an image is loaded. Once an image is placed in the image cache, it can always be displayed, even if the value of *max-image-size* is subsequently changed (see Section 11.16.13 [Image Cache], page 178).

### 11.16.12 Animated Images

Some image files can contain more than one image. This can be used to create animation. Currently, Emacs only supports animated GIF files. The following functions related to animated images are available.

`image-animated-p` *image* [Function]
: This function returns non-`nil` if *image* can be animated. The actual return value is a cons `(`*`nimages`* `.` *`delay`*`)`, where *nimages* is the number of frames and *delay* is the delay in seconds between them.

`image-animate` *image* **&optional** *index limit* [Function]
: This function animates *image*. The optional integer *index* specifies the frame from which to start (default 0). The optional argument *limit* controls the length of the animation. If omitted or `nil`, the image animates once only; if `t` it loops forever; if a number animation stops after that many seconds.

Animation operates by means of a timer. Note that Emacs imposes a minimum frame delay of 0.01 seconds.

`image-animate-timer` *image* [Function]
: This function returns the timer responsible for animating *image*, if there is one.

### 11.16.13 Image Cache

Emacs caches images so that it can display them again more efficiently. When Emacs displays an image, it searches the image cache for an existing image specification `equal` to the desired specification. If a match is found, the image is displayed from the cache. Otherwise, Emacs loads the image normally.

`image-flush` *spec* **&optional** *frame* [Function]
: This function removes the image with specification *spec* from the image cache of frame *frame*. Image specifications are compared using `equal`. If *frame* is `nil`, it defaults to the selected frame. If *frame* is `t`, the image is flushed on all existing frames.

In Emacs's current implementation, each graphical terminal possesses an image cache, which is shared by all the frames on that terminal (see Section 18.2 [Multiple Terminals], page 342). Thus, refreshing an image in one frame also refreshes it in all other frames on the same terminal.

One use for `image-flush` is to tell Emacs about a change in an image file. If an image specification contains a `:file` property, the image is cached based on the file's contents when the image is first displayed. Even if the file subsequently changes, Emacs continues displaying the old version of the image. Calling `image-flush` flushes the image from the cache, forcing Emacs to re-read the file the next time it needs to display that image.

Another use for `image-flush` is for memory conservation. If your Lisp program creates a large number of temporary images over a period much shorter than `image-cache-eviction-delay` (see below), you can opt to flush unused images yourself, instead of waiting for Emacs to do it automatically.

`clear-image-cache` **&optional** *filter* [Function]

This function clears an image cache, removing all the images stored in it. If *filter* is omitted or `nil`, it clears the cache for the selected frame. If *filter* is a frame, it clears the cache for that frame. If *filter* is `t`, all image caches are cleared. Otherwise, *filter* is taken to be a file name, and all images associated with that file name are removed from all image caches.

If an image in the image cache has not been displayed for a specified period of time, Emacs removes it from the cache and frees the associated memory.

`image-cache-eviction-delay` [Variable]

This variable specifies the number of seconds an image can remain in the cache without being displayed. When an image is not displayed for this length of time, Emacs removes it from the image cache.

Under some circumstances, if the number of images in the cache grows too large, the actual eviction delay may be shorter than this.

If the value is `nil`, Emacs does not remove images from the cache except when you explicitly clear it. This mode can be useful for debugging.

## 11.17 Buttons

The Button package defines functions for inserting and manipulating *buttons* that can be activated with the mouse or via keyboard commands. These buttons are typically used for various kinds of hyperlinks.

A button is essentially a set of text or overlay properties, attached to a stretch of text in a buffer. These properties are called *button properties*. One of these properties, the *action property*, specifies a function which is called when the user invokes the button using the keyboard or the mouse. The action function may examine the button and use its other properties as desired.

In some ways, the Button package duplicates the functionality in the Widget package. See Section "Introduction" in *The Emacs Widget Library*. The advantage of the Button package is that it is faster, smaller, and simpler to program. From the point of view of the user, the interfaces produced by the two packages are very similar.

### 11.17.1 Button Properties

Each button has an associated list of properties defining its appearance and behavior, and other arbitrary properties may be used for application specific purposes. The following properties have special meaning to the Button package:

`action` The function to call when the user invokes the button, which is passed the single argument *button*. By default this is `ignore`, which does nothing.

`mouse-action`
This is similar to `action`, and when present, will be used instead of `action` for button invocations resulting from mouse-clicks (instead of the user hitting RET). If not present, mouse-clicks use `action` instead.

`face` This is an Emacs face controlling how buttons of this type are displayed; by default this is the `button` face.

`mouse-face`
This is an additional face which controls appearance during mouse-overs (merged with the usual button face); by default this is the usual Emacs `highlight` face.

`keymap` The button's keymap, defining bindings active within the button region. By default this is the usual button region keymap, stored in the variable `button-map`, which defines RET and MOUSE-2 to invoke the button.

`type` The button type. See Section 11.17.2 [Button Types], page 180.

`help-echo`
A string displayed by the Emacs tool-tip help system; by default, `"mouse-2, RET: Push this button"`.

`follow-link`
The follow-link property, defining how a MOUSE-1 click behaves on this button, See Section 22.19.8 [Clickable Text], page 502.

`button` All buttons have a non-`nil` `button` property, which may be useful in finding regions of text that comprise buttons (which is what the standard button functions do).

There are other properties defined for the regions of text in a button, but these are not generally interesting for typical uses.

### 11.17.2 Button Types

Every button has a *button type*, which defines default values for the button's properties. Button types are arranged in a hierarchy, with specialized types inheriting from more general types, so that it's easy to define special-purpose types of buttons for specific tasks.

`define-button-type` *name* **&rest** *properties* [Function]
Define a 'button type' called *name* (a symbol). The remaining arguments form a sequence of *property value* pairs, specifying default property values for buttons with this type (a button's type may be set by giving it a `type` property when creating the button, using the `:type` keyword argument).

In addition, the keyword argument `:supertype` may be used to specify a button-type from which *name* inherits its default property values. Note that this inheritance happens only when *name* is defined; subsequent changes to a supertype are not reflected in its subtypes.

Using `define-button-type` to define default properties for buttons is not necessary—buttons without any specified type use the built-in button-type `button`—but it is encouraged, since doing so usually makes the resulting code clearer and more efficient.

### 11.17.3 Making Buttons

Buttons are associated with a region of text, using an overlay or text properties to hold button-specific information, all of which are initialized from the button's type (which defaults to the built-in button type `button`). Like all Emacs text, the appearance of the button is governed by the `face` property; by default (via the `face` property inherited from the `button` button-type) this is a simple underline, like a typical web-page link.

For convenience, there are two sorts of button-creation functions, those that add button properties to an existing region of a buffer, called `make-...button`, and those that also insert the button text, called `insert-...button`.

The button-creation functions all take the `&rest` argument *properties*, which should be a sequence of *property value* pairs, specifying properties to add to the button; see Section 11.17.1 [Button Properties], page 179. In addition, the keyword argument `:type` may be used to specify a button-type from which to inherit other properties; see Section 11.17.2 [Button Types], page 180. Any properties not explicitly specified during creation will be inherited from the button's type (if the type defines such a property).

The following functions add a button using an overlay (see Section 11.9 [Overlays], page 128) to hold the button properties:

`make-button` *beg end* **&rest** *properties* [Function]
: This makes a button from *beg* to *end* in the current buffer, and returns it.

`insert-button` *label* **&rest** *properties* [Function]
: This insert a button with the label *label* at point, and returns it.

The following functions are similar, but using text properties (see Section 22.19 [Text Properties], page 489) to hold the button properties. Such buttons do not add markers to the buffer, so editing in the buffer does not slow down if there is an extremely large numbers of buttons. However, if there is an existing face text property on the text (e.g., a face assigned by Font Lock mode), the button face may not be visible. Both of these functions return the starting position of the new button.

`make-text-button` *beg end* **&rest** *properties* [Function]
: This makes a button from *beg* to *end* in the current buffer, using text properties.

`insert-text-button` *label* **&rest** *properties* [Function]
: This inserts a button with the label *label* at point, using text properties.

### 11.17.4 Manipulating Buttons

These are functions for getting and setting properties of buttons. Often these are used by a button's invocation function to determine what to do.

Where a *button* parameter is specified, it means an object referring to a specific button, either an overlay (for overlay buttons), or a buffer-position or marker (for text property buttons). Such an object is passed as the first argument to a button's invocation function when it is invoked.

`button-start` *button* [Function]
Return the position at which *button* starts.

`button-end` *button* [Function]
Return the position at which *button* ends.

`button-get` *button prop* [Function]
Get the property of button *button* named *prop*.

`button-put` *button prop val* [Function]
Set *button*'s *prop* property to *val*.

`button-activate` *button* **&optional** *use-mouse-action* [Function]
Call *button*'s `action` property (i.e., invoke it). If *use-mouse-action* is non-`nil`, try to invoke the button's `mouse-action` property instead of `action`; if the button has no `mouse-action` property, use `action` as normal.

`button-label` *button* [Function]
Return *button*'s text label.

`button-type` *button* [Function]
Return *button*'s button-type.

`button-has-type-p` *button type* [Function]
Return `t` if *button* has button-type *type*, or one of *type*'s subtypes.

`button-at` *pos* [Function]
Return the button at position *pos* in the current buffer, or `nil`. If the button at *pos* is a text property button, the return value is a marker pointing to *pos*.

`button-type-put` *type prop val* [Function]
Set the button-type *type*'s *prop* property to *val*.

`button-type-get` *type prop* [Function]
Get the property of button-type *type* named *prop*.

`button-type-subtype-p` *type supertype* [Function]
Return `t` if button-type *type* is a subtype of *supertype*.

### 11.17.5 Button Buffer Commands

These are commands and functions for locating and operating on buttons in an Emacs buffer.

`push-button` is the command that a user uses to actually 'push' a button, and is bound by default in the button itself to RET and to MOUSE-2 using a local keymap in the button's overlay or text properties. Commands that are useful outside the buttons itself, such as `forward-button` and `backward-button` are additionally available in the keymap stored in `button-buffer-map`; a mode which uses buttons may want to use `button-buffer-map` as a parent keymap for its keymap.

If the button has a non-`nil` `follow-link` property, and *mouse-1-click-follows-link* is set, a quick MOUSE-1 click will also activate the `push-button` command. See Section 22.19.8 [Clickable Text], page 502.

`push-button` **&optional** *pos use-mouse-action* [Command]
: Perform the action specified by a button at location *pos*. *pos* may be either a buffer position or a mouse-event. If *use-mouse-action* is non-`nil`, or *pos* is a mouse-event (see Section 2.7.3 [Mouse Events], page 25), try to invoke the button's `mouse-action` property instead of `action`; if the button has no `mouse-action` property, use `action` as normal. *pos* defaults to point, except when `push-button` is invoked interactively as the result of a mouse-event, in which case, the mouse event's position is used. If there's no button at *pos*, do nothing and return `nil`, otherwise return `t`.

`forward-button` *n* **&optional** *wrap display-message* [Command]
: Move to the *n*th next button, or *n*th previous button if *n* is negative. If *n* is zero, move to the start of any button at point. If *wrap* is non-`nil`, moving past either end of the buffer continues from the other end. If *display-message* is non-`nil`, the button's help-echo string is displayed. Any button with a non-`nil` `skip` property is skipped over. Returns the button found.

`backward-button` *n* **&optional** *wrap display-message* [Command]
: Move to the *n*th previous button, or *n*th next button if *n* is negative. If *n* is zero, move to the start of any button at point. If *wrap* is non-`nil`, moving past either end of the buffer continues from the other end. If *display-message* is non-`nil`, the button's help-echo string is displayed. Any button with a non-`nil` `skip` property is skipped over. Returns the button found.

`next-button` *pos* **&optional** *count-current* [Function]
`previous-button` *pos* **&optional** *count-current* [Function]
: Return the next button after (for `next-button` or before (for `previous-button`) position *pos* in the current buffer. If *count-current* is non-`nil`, count any button at *pos* in the search, instead of starting at the next button.

## 11.18 Abstract Display

The Ewoc package constructs buffer text that represents a structure of Lisp objects, and updates the text to follow changes in that structure. This is like the "view" component in the "model/view/controller" design paradigm.

An *ewoc* is a structure that organizes information required to construct buffer text that represents certain Lisp data. The buffer text of the ewoc has three parts, in order: first, fixed *header* text; next, textual descriptions of a series of data elements (Lisp objects that you specify); and last, fixed *footer* text. Specifically, an ewoc contains information on:

- The buffer which its text is generated in.
- The text's start position in the buffer.
- The header and footer strings.
- A doubly-linked chain of *nodes*, each of which contains:
  - A *data element*, a single Lisp object.
  - Links to the preceding and following nodes in the chain.
- A *pretty-printer* function which is responsible for inserting the textual representation of a data element value into the current buffer.

Typically, you define an ewoc with `ewoc-create`, and then pass the resulting ewoc structure to other functions in the Ewoc package to build nodes within it, and display it in the buffer. Once it is displayed in the buffer, other functions determine the correspondence between buffer positions and nodes, move point from one node's textual representation to another, and so forth. See Section 11.18.1 [Abstract Display Functions], page 184.

A node *encapsulates* a data element much the way a variable holds a value. Normally, encapsulation occurs as a part of adding a node to the ewoc. You can retrieve the data element value and place a new value in its place, like so:

```
(ewoc-data node)
⇒ value

(ewoc-set-data node new-value)
⇒ new-value
```

You can also use, as the data element value, a Lisp object (list or vector) that is a container for the "real" value, or an index into some other structure. The example (see Section 11.18.2 [Abstract Display Example], page 186) uses the latter approach.

When the data changes, you will want to update the text in the buffer. You can update all nodes by calling `ewoc-refresh`, or just specific nodes using `ewoc-invalidate`, or all nodes satisfying a predicate using `ewoc-map`. Alternatively, you can delete invalid nodes using `ewoc-delete` or `ewoc-filter`, and add new nodes in their place. Deleting a node from an ewoc deletes its associated textual description from buffer, as well.

### 11.18.1 Abstract Display Functions

In this subsection, *ewoc* and *node* stand for the structures described above (see Section 11.18 [Abstract Display], page 183), while *data* stands for an arbitrary Lisp object used as a data element.

`ewoc-create` *pretty-printer* **&optional** *header footer nosep* [Function]

This constructs and returns a new ewoc, with no nodes (and thus no data elements). *pretty-printer* should be a function that takes one argument, a data element of the sort you plan to use in this ewoc, and inserts its textual description at point using `insert` (and never `insert-before-markers`, because that would interfere with the Ewoc package's internal mechanisms).

Normally, a newline is automatically inserted after the header, the footer and every node's textual description. If *nosep* is non-`nil`, no newline is inserted. This may be useful for displaying an entire ewoc on a single line, for example, or for making nodes "invisible" by arranging for *pretty-printer* to do nothing for those nodes.

An ewoc maintains its text in the buffer that is current when you create it, so switch to the intended buffer before calling `ewoc-create`.

`ewoc-buffer` *ewoc* [Function]

This returns the buffer where *ewoc* maintains its text.

`ewoc-get-hf` *ewoc* [Function]

This returns a cons cell (*header* . *footer*) made from *ewoc*'s header and footer.

`ewoc-set-hf` *ewoc header footer* [Function]
This sets the header and footer of *ewoc* to the strings *header* and *footer*, respectively.

`ewoc-enter-first` *ewoc data* [Function]
`ewoc-enter-last` *ewoc data* [Function]
These add a new node encapsulating *data*, putting it, respectively, at the beginning or end of *ewoc*'s chain of nodes.

`ewoc-enter-before` *ewoc node data* [Function]
`ewoc-enter-after` *ewoc node data* [Function]
These add a new node encapsulating *data*, adding it to *ewoc* before or after *node*, respectively.

`ewoc-prev` *ewoc node* [Function]
`ewoc-next` *ewoc node* [Function]
These return, respectively, the previous node and the next node of *node* in *ewoc*.

`ewoc-nth` *ewoc n* [Function]
This returns the node in *ewoc* found at zero-based index *n*. A negative *n* means count from the end. `ewoc-nth` returns `nil` if *n* is out of range.

`ewoc-data` *node* [Function]
This extracts the data encapsulated by *node* and returns it.

`ewoc-set-data` *node data* [Function]
This sets the data encapsulated by *node* to *data*.

`ewoc-locate` *ewoc* **&optional** *pos guess* [Function]
This determines the node in *ewoc* which contains point (or *pos* if specified), and returns that node. If *ewoc* has no nodes, it returns `nil`. If *pos* is before the first node, it returns the first node; if *pos* is after the last node, it returns the last node. The optional third arg *guess* should be a node that is likely to be near *pos*; this doesn't alter the result, but makes the function run faster.

`ewoc-location` *node* [Function]
This returns the start position of *node*.

`ewoc-goto-prev` *ewoc arg* [Function]
`ewoc-goto-next` *ewoc arg* [Function]
These move point to the previous or next, respectively, *arg*th node in *ewoc*. `ewoc-goto-prev` does not move if it is already at the first node or if *ewoc* is empty, whereas `ewoc-goto-next` moves past the last node, returning `nil`. Excepting this special case, these functions return the node moved to.

`ewoc-goto-node` *ewoc node* [Function]
This moves point to the start of *node* in *ewoc*.

`ewoc-refresh` *ewoc* [Function]
This function regenerates the text of *ewoc*. It works by deleting the text between the header and the footer, i.e., all the data elements' representations, and then calling the pretty-printer function for each node, one by one, in order.

`ewoc-invalidate` *ewoc* **&rest** *nodes* [Function]
This is similar to `ewoc-refresh`, except that only *nodes* in *ewoc* are updated instead of the entire set.

`ewoc-delete` *ewoc* **&rest** *nodes* [Function]
This deletes each node in *nodes* from *ewoc*.

`ewoc-filter` *ewoc predicate* **&rest** *args* [Function]
This calls *predicate* for each data element in *ewoc* and deletes those nodes for which *predicate* returns `nil`. Any *args* are passed to *predicate*.

`ewoc-collect` *ewoc predicate* **&rest** *args* [Function]
This calls *predicate* for each data element in *ewoc* and returns a list of those elements for which *predicate* returns non-`nil`. The elements in the list are ordered as in the buffer. Any *args* are passed to *predicate*.

`ewoc-map` *map-function ewoc* **&rest** *args* [Function]
This calls *map-function* for each data element in *ewoc* and updates those nodes for which *map-function* returns non-`nil`. Any *args* are passed to *map-function*.

### 11.18.2 Abstract Display Example

Here is a simple example using functions of the ewoc package to implement a "color components display", an area in a buffer that represents a vector of three integers (itself representing a 24-bit RGB value) in various ways.

```
(setq colorcomp-ewoc nil
      colorcomp-data nil
      colorcomp-mode-map nil
      colorcomp-labels ["Red" "Green" "Blue"])

(defun colorcomp-pp (data)
  (if data
      (let ((comp (aref colorcomp-data data)))
        (insert (aref colorcomp-labels data) "\t: #x"
                (format "%02X" comp) " "
                (make-string (ash comp -2) ?#) "\n"))
    (let ((cstr (format "#%02X%02X%02X"
                        (aref colorcomp-data 0)
                        (aref colorcomp-data 1)
                        (aref colorcomp-data 2)))
          (samp " (sample text) "))
      (insert "Color\t: "
              (propertize samp 'face
                          `(foreground-color . ,cstr))
              (propertize samp 'face
                          `(background-color . ,cstr))
              "\n"))))

(defun colorcomp (color)
```

```
  "Allow fiddling with COLOR in a new buffer.
The buffer is in Color Components mode."
  (interactive "sColor (name or #RGB or #RRGGBB): ")
  (when (string= "" color)
    (setq color "green"))
  (unless (color-values color)
    (error "No such color: %S" color))
  (switch-to-buffer
   (generate-new-buffer (format "originally: %s" color)))
  (kill-all-local-variables)
  (setq major-mode 'colorcomp-mode
        mode-name "Color Components")
  (use-local-map colorcomp-mode-map)
  (erase-buffer)
  (buffer-disable-undo)
  (let ((data (apply 'vector (mapcar (lambda (n) (ash n -8))
                                     (color-values color))))
        (ewoc (ewoc-create 'colorcomp-pp
                           "\nColor Components\n\n"
                           (substitute-command-keys
                            "\n\\{colorcomp-mode-map}"))))
    (set (make-local-variable 'colorcomp-data) data)
    (set (make-local-variable 'colorcomp-ewoc) ewoc)
    (ewoc-enter-last ewoc 0)
    (ewoc-enter-last ewoc 1)
    (ewoc-enter-last ewoc 2)
    (ewoc-enter-last ewoc nil)))
```

This example can be extended to be a "color selection widget" (in other words, the controller part of the "model/view/controller" design paradigm) by defining commands to modify `colorcomp-data` and to "finish" the selection process, and a keymap to tie it all together conveniently.

```
(defun colorcomp-mod (index limit delta)
  (let ((cur (aref colorcomp-data index)))
    (unless (= limit cur)
      (aset colorcomp-data index (+ cur delta)))
    (ewoc-invalidate
     colorcomp-ewoc
     (ewoc-nth colorcomp-ewoc index)
     (ewoc-nth colorcomp-ewoc -1))))

(defun colorcomp-R-more () (interactive) (colorcomp-mod 0 255 1))
(defun colorcomp-G-more () (interactive) (colorcomp-mod 1 255 1))
(defun colorcomp-B-more () (interactive) (colorcomp-mod 2 255 1))
(defun colorcomp-R-less () (interactive) (colorcomp-mod 0 0 -1))
(defun colorcomp-G-less () (interactive) (colorcomp-mod 1 0 -1))
(defun colorcomp-B-less () (interactive) (colorcomp-mod 2 0 -1))

(defun colorcomp-copy-as-kill-and-exit ()
  "Copy the color components into the kill ring and kill the buffer.
The string is formatted #RRGGBB (hash followed by six hex digits)."
```

```
  (interactive)
  (kill-new (format "#%02X%02X%02X"
                    (aref colorcomp-data 0)
                    (aref colorcomp-data 1)
                    (aref colorcomp-data 2)))
  (kill-buffer nil))

(setq colorcomp-mode-map
      (let ((m (make-sparse-keymap)))
        (suppress-keymap m)
        (define-key m "i" 'colorcomp-R-less)
        (define-key m "o" 'colorcomp-R-more)
        (define-key m "k" 'colorcomp-G-less)
        (define-key m "l" 'colorcomp-G-more)
        (define-key m "," 'colorcomp-B-less)
        (define-key m "." 'colorcomp-B-more)
        (define-key m " " 'colorcomp-copy-as-kill-and-exit)
        m))
```

Note that we never modify the data in each node, which is fixed when the ewoc is created to be either `nil` or an index into the vector `colorcomp-data`, the actual color components.

## 11.19 Blinking Parentheses

This section describes the mechanism by which Emacs shows a matching open parenthesis when the user inserts a close parenthesis.

`blink-paren-function` [Variable]

The value of this variable should be a function (of no arguments) to be called whenever a character with close parenthesis syntax is inserted. The value of `blink-paren-function` may be `nil`, in which case nothing is done.

`blink-matching-paren` [User Option]

If this variable is `nil`, then `blink-matching-open` does nothing.

`blink-matching-paren-distance` [User Option]

This variable specifies the maximum distance to scan for a matching parenthesis before giving up.

`blink-matching-delay` [User Option]

This variable specifies the number of seconds for the cursor to remain at the matching parenthesis. A fraction of a second often gives good results, but the default is 1, which works on all systems.

`blink-matching-open` [Command]

This function is the default value of `blink-paren-function`. It assumes that point follows a character with close parenthesis syntax and moves the cursor momentarily to the matching opening character. If that character is not already on the screen, it displays the character's context in the echo area. To avoid long delays, this function does not search farther than `blink-matching-paren-distance` characters.

Here is an example of calling this function explicitly.

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of sexp before point."
  (interactive)
```

```
(let ((blink-matching-paren-distance
       (buffer-size))
      (blink-matching-paren t))
  (blink-matching-open)))
```

## 11.20 Character Display

This section describes how characters are actually displayed by Emacs. Typically, a character is displayed as a *glyph* (a graphical symbol which occupies one character position on the screen), whose appearance corresponds to the character itself. For example, the character '`a`' (character code 97) is displayed as '`a`'. Some characters, however, are displayed specially. For example, the formfeed character (character code 12) is usually displayed as a sequence of two glyphs, '`^L`', while the newline character (character code 10) starts a new screen line.

You can modify how each character is displayed by defining a *display table*, which maps each character code into a sequence of glyphs. See Section 11.20.2 [Display Tables], page 190.

### 11.20.1 Usual Display Conventions

Here are the conventions for displaying each character code (in the absence of a display table, which can override these conventions).

- The *printable ASCII characters*, character codes 32 through 126 (consisting of numerals, English letters, and symbols like '`#`') are displayed literally.
- The tab character (character code 9) displays as whitespace stretching up to the next tab stop column. See Section "Text Display" in *The GNU Emacs Manual*. The variable `tab-width` controls the number of spaces per tab stop (see below).
- The newline character (character code 10) has a special effect: it ends the preceding line and starts a new line.
- The non-printable *ASCII control characters*—character codes 0 through 31, as well as the DEL character (character code 127)—display in one of two ways according to the variable `ctl-arrow`. If this variable is non-`nil` (the default), these characters are displayed as sequences of two glyphs, where the first glyph is '`^`' (a display table can specify a glyph to use instead of '`^`'); e.g., the DEL character is displayed as '`^?`'.

  If `ctl-arrow` is `nil`, these characters are displayed as octal escapes (see below).

  This rule also applies to carriage return (character code 13), if that character appears in the buffer. But carriage returns usually do not appear in buffer text; they are eliminated as part of end-of-line conversion (see ⟨undefined⟩ [Coding System Basics], page ⟨undefined⟩).
- *Raw bytes* are non-ASCII characters with codes 128 through 255 (see ⟨undefined⟩ [Text Representations], page ⟨undefined⟩). These characters display as *octal escapes*: sequences of four glyphs, where the first glyph is the ASCII code for '\', and the others are digit characters representing the character code in octal. (A display table can specify a glyph to use instead of '\'.)
- Each non-ASCII character with code above 255 is displayed literally, if the terminal supports it. If the terminal does not support it, the character is said to be *glyphless*, and it is usually displayed using a placeholder glyph. For example, if a graphical terminal has no font for a character, Emacs usually displays a box containing the character code in hexadecimal. See Section 11.20.5 [Glyphless Chars], page 192.

The above display conventions apply even when there is a display table, for any character whose entry in the active display table is `nil`. Thus, when you set up a display table, you need only specify the characters for which you want special behavior.

The following variables affect how certain characters are displayed on the screen. Since they change the number of columns the characters occupy, they also affect the indentation functions. They also affect how the mode line is displayed; if you want to force redisplay of the mode line using the new values, call the function `force-mode-line-update` (see Section 20.4 [Mode Line Format], page 418).

`ctl-arrow` [User Option]

This buffer-local variable controls how control characters are displayed. If it is non-`nil`, they are displayed as a caret followed by the character: '`^A`'. If it is `nil`, they are displayed as octal escapes: a backslash followed by three octal digits, as in '`\001`'.

`tab-width` [User Option]

The value of this buffer-local variable is the spacing between tab stops used for displaying tab characters in Emacs buffers. The value is in units of columns, and the default is 8. Note that this feature is completely independent of the user-settable tab stops used by the command `tab-to-tab-stop`. See Section 22.17.5 [Indent Tabs], page 487.

### 11.20.2 Display Tables

A display table is a special-purpose char-table (see ⟨undefined⟩ [Char-Tables], page ⟨undefined⟩), with `display-table` as its subtype, which is used to override the usual character display conventions. This section describes how to make, inspect, and assign elements to a display table object.

`make-display-table` [Function]

This creates and returns a display table. The table initially has `nil` in all elements.

The ordinary elements of the display table are indexed by character codes; the element at index *c* says how to display the character code *c*. The value should be `nil` (which means to display the character *c* according to the usual display conventions; see Section 11.20.1 [Usual Display], page 189), or a vector of glyph codes (which means to display the character *c* as those glyphs; see Section 11.20.4 [Glyphs], page 192).

**Warning:** if you use the display table to change the display of newline characters, the whole buffer will be displayed as one long "line".

The display table also has six "extra slots" which serve special purposes. Here is a table of their meanings; `nil` in any slot means to use the default for that slot, as stated below.

0 The glyph for the end of a truncated screen line (the default for this is '`$`'). See Section 11.20.4 [Glyphs], page 192. On graphical terminals, Emacs uses arrows in the fringes to indicate truncation, so the display table has no effect.

1 The glyph for the end of a continued line (the default is '\'). On graphical terminals, Emacs uses curved arrows in the fringes to indicate continuation, so the display table has no effect.

2 The glyph for indicating a character displayed as an octal character code (the default is '\').

3 The glyph for indicating a control character (the default is '^').

4 A vector of glyphs for indicating the presence of invisible lines (the default is '...'). See Section 11.7 [Selective Display], page 124.

5 The glyph used to draw the border between side-by-side windows (the default is '|'). See Section 17.5 [Splitting Windows], page 297. This takes effect only when there are no scroll bars; if scroll bars are supported and in use, a scroll bar separates the two windows.

For example, here is how to construct a display table that mimics the effect of setting `ctl-arrow` to a non-`nil` value (see Section 11.20.4 [Glyphs], page 192, for the function `make-glyph-code`):

```
(setq disptab (make-display-table))
(dotimes (i 32)
  (or (= i ?\t)
      (= i ?\n)
      (aset disptab i
            (vector (make-glyph-code ?^ 'escape-glyph)
                    (make-glyph-code (+ i 64) 'escape-glyph)))))
(aset disptab 127
      (vector (make-glyph-code ?^ 'escape-glyph)
              (make-glyph-code ?? 'escape-glyph))))
```

`display-table-slot` *display-table slot* [Function]
This function returns the value of the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

`set-display-table-slot` *display-table slot value* [Function]
This function stores *value* in the extra slot *slot* of *display-table*. The argument *slot* may be a number from 0 to 5 inclusive, or a slot name (symbol). Valid symbols are `truncation`, `wrap`, `escape`, `control`, `selective-display`, and `vertical-border`.

`describe-display-table` *display-table* [Function]
This function displays a description of the display table *display-table* in a help buffer.

`describe-current-display-table` [Command]
This command displays a description of the current display table in a help buffer.

### 11.20.3 Active Display Table

Each window can specify a display table, and so can each buffer. The window's display table, if there is one, takes precedence over the buffer's display table. If neither exists, Emacs tries to use the standard display table; if that is `nil`, Emacs uses the usual character display conventions (see Section 11.20.1 [Usual Display], page 189).

Note that display tables affect how the mode line is displayed, so if you want to force redisplay of the mode line using a new display table, call `force-mode-line-update` (see Section 20.4 [Mode Line Format], page 418).

`window-display-table` **&optional** *window* [Function]
This function returns *window*'s display table, or `nil` if there is none. The default for *window* is the selected window.

`set-window-display-table` *window table* [Function]
This function sets the display table of *window* to *table*. The argument *table* should be either a display table or `nil`.

`buffer-display-table` [Variable]
This variable is automatically buffer-local in all buffers; its value specifies the buffer's display table. If it is `nil`, there is no buffer display table.

`standard-display-table` [Variable]
The value of this variable is the standard display table, which is used when Emacs is displaying a buffer in a window with neither a window display table nor a buffer display table defined. Its default is `nil`.

The '`disp-table`' library defines several functions for changing the standard display table.

### 11.20.4 Glyphs

A *glyph* is a graphical symbol which occupies a single character position on the screen. Each glyph is represented in Lisp as a *glyph code*, which specifies a character and optionally a face to display it in (see Section 11.12 [Faces], page 137). The main use of glyph codes is as the entries of display tables (see Section 11.20.2 [Display Tables], page 190). The following functions are used to manipulate glyph codes:

`make-glyph-code` *char* **&optional** *face* [Function]
This function returns a glyph code representing char *char* with face *face*. If *face* is omitted or `nil`, the glyph uses the default face; in that case, the glyph code is an integer. If *face* is non-`nil`, the glyph code is not necessarily an integer object.

`glyph-char` *glyph* [Function]
This function returns the character of glyph code *glyph*.

`glyph-face` *glyph* [Function]
This function returns face of glyph code *glyph*, or `nil` if *glyph* uses the default face.

### 11.20.5 Glyphless Character Display

*Glyphless characters* are characters which are displayed in a special way, e.g., as a box containing a hexadecimal code, instead of being displayed literally. These include characters which are explicitly defined to be glyphless, as well as characters for which there is no available font (on a graphical display), and characters which cannot be encoded by the terminal's coding system (on a text terminal).

`glyphless-char-display` [Variable]
The value of this variable is a char-table which defines glyphless characters and how they are displayed. Each entry must be one of the following display methods:

`nil` Display the character in the usual way.

`zero-width`
: Don't display the character.

`thin-space`
: Display a thin space, 1-pixel wide on graphical displays, or 1-character wide on text terminals.

`empty-box`
: Display an empty box.

`hex-code` Display a box containing the Unicode codepoint of the character, in hexadecimal notation.

an ASCII string
: Display a box containing that string.

a cons cell (*graphical* . *text*)
: Display with *graphical* on graphical displays, and with *text* on text terminals. Both *graphical* and *text* must be one of the display methods described above.

The `thin-space`, `empty-box`, `hex-code`, and ASCII string display methods are drawn with the `glyphless-char` face.

The char-table has one extra slot, which determines how to display any character that cannot be displayed with any available font, or cannot be encoded by the terminal's coding system. Its value should be one of the above display methods, except `zero-width` or a cons cell.

If a character has a non-`nil` entry in an active display table, the display table takes effect; in this case, Emacs does not consult `glyphless-char-display` at all.

`glyphless-char-display-control` [User Option]

This user option provides a convenient way to set `glyphless-char-display` for groups of similar characters. Do not set its value directly from Lisp code; the value takes effect only via a custom `:set` function (see ⟨undefined⟩ [Variable Definitions], page ⟨undefined⟩), which updates `glyphless-char-display`.

Its value should be an alist of elements (*group* . *method*), where *group* is a symbol specifying a group of characters, and *method* is a symbol specifying how to display them.

*group* should be one of the following:

`c0-control`
: ASCII control characters `U+0000` to `U+001F`, excluding the newline and tab characters (normally displayed as escape sequences like '`^A`'; see Section "How Text Is Displayed" in *The GNU Emacs Manual*).

`c1-control`
: Non-ASCII, non-printing characters `U+0080` to `U+009F` (normally displayed as octal escape sequences like '`\230`').

`format-control`
: Characters of Unicode General Category 'Cf', such as '`U+200E`' (Left-to-Right Mark), but excluding characters that have graphic images, such as '`U+00AD`' (Soft Hyphen).

`no-font` Characters for there is no suitable font, or which cannot be encoded by the terminal's coding system.

The *method* symbol should be one of `zero-width`, `thin-space`, `empty-box`, or `hex-code`. These have the same meanings as in `glyphless-char-display`, above.

## 11.21 Beeping

This section describes how to make Emacs ring the bell (or blink the screen) to attract the user's attention. Be conservative about how often you do this; frequent bells can become irritating. Also be careful not to use just beeping when signaling an error is more appropriate (see ⟨undefined⟩ [Errors], page ⟨undefined⟩).

`ding` **&optional** *do-not-terminate* [Function]
This function beeps, or flashes the screen (see `visible-bell` below). It also terminates any keyboard macro currently executing unless *do-not-terminate* is non-`nil`.

`beep` **&optional** *do-not-terminate* [Function]
This is a synonym for `ding`.

`visible-bell` [User Option]
This variable determines whether Emacs should flash the screen to represent a bell. Non-`nil` means yes, `nil` means no. This is effective on graphical displays, and on text terminals provided the terminal's Termcap entry defines the visible bell capability (‘`vb`’).

`ring-bell-function` [Variable]
If this is non-`nil`, it specifies how Emacs should "ring the bell". Its value should be a function of no arguments. If this is non-`nil`, it takes precedence over the `visible-bell` variable.

## 11.22 Window Systems

Emacs works with several window systems, most notably the X Window System. Both Emacs and X use the term "window", but use it differently. An Emacs frame is a single window as far as X is concerned; the individual Emacs windows are not known to X at all.

`window-system` [Variable]
This terminal-local variable tells Lisp programs what window system Emacs is using for displaying the frame. The possible values are

`x` Emacs is displaying the frame using X.

`w32` Emacs is displaying the frame using native MS-Windows GUI.

`ns` Emacs is displaying the frame using the Nextstep interface (used on GNUstep and Mac OS X).

`pc` Emacs is displaying the frame using MS-DOS direct screen writes.

`nil` Emacs is displaying the frame on a character-based terminal.

`initial-window-system` [Variable]
: This variable holds the value of `window-system` used for the first frame created by Emacs during startup. (When Emacs is invoked with the '`--daemon`' option, it does not create any initial frames, so `initial-window-system` is `nil`. See Section "Initial Options" in *The GNU Emacs Manual*.)

`window-system` **&optional** *frame* [Function]
: This function returns a symbol whose name tells what window system is used for displaying *frame* (which defaults to the currently selected frame). The list of possible symbols it returns is the same one documented for the variable `window-system` above.

Do *not* use `window-system` and `initial-window-system` as predicates or boolean flag variables, if you want to write code that works differently on text terminals and graphic displays. That is because `window-system` is not a good indicator of Emacs capabilities on a given display type. Instead, use `display-graphic-p` or any of the other `display-*-p` predicates described in Section 18.23 [Display Feature Testing], page 370.

`window-setup-hook` [Variable]
: This variable is a normal hook which Emacs runs after handling the initialization files. Emacs runs this hook after it has completed loading your init file, the default initialization file (if any), and the terminal-specific Lisp code, and running the hook `term-setup-hook`.

  This hook is used for internal purposes: setting up communication with the window system, and creating the initial window. Users should not interfere with it.

## 11.23 Bidirectional Display

Emacs can display text written in scripts, such as Arabic, Farsi, and Hebrew, whose natural ordering for horizontal text display runs from right to left. Furthermore, segments of Latin script and digits embedded in right-to-left text are displayed left-to-right, while segments of right-to-left script embedded in left-to-right text (e.g., Arabic or Hebrew text in comments or strings in a program source file) are appropriately displayed right-to-left. We call such mixtures of left-to-right and right-to-left text *bidirectional text*. This section describes the facilities and options for editing and displaying bidirectional text.

Text is stored in Emacs buffers and strings in *logical* (or *reading*) order, i.e., the order in which a human would read each character. In right-to-left and bidirectional text, the order in which characters are displayed on the screen (called *visual order*) is not the same as logical order; the characters' screen positions do not increase monotonically with string or buffer position. In performing this *bidirectional reordering*, Emacs follows the Unicode Bidirectional Algorithm (a.k.a. UBA), which is described in Annex #9 of the Unicode standard (`http://www.unicode.org/reports/tr9/`). Emacs provides a "Full Bidirectionality" class implementation of the UBA.

`bidi-display-reordering` [Variable]
: If the value of this buffer-local variable is non-`nil` (the default), Emacs performs bidirectional reordering for display. The reordering affects buffer text, as well as display strings and overlay strings from text and overlay properties in the buffer (see Section 11.9.2 [Overlay Properties], page 131, and see Section 11.15 [Display

Property], page 163). If the value is `nil`, Emacs does not perform bidirectional reordering in the buffer.

The default value of `bidi-display-reordering` controls the reordering of strings which are not directly supplied by a buffer, including the text displayed in mode lines (see Section 20.4 [Mode Line Format], page 418) and header lines (see Section 20.4.7 [Header Lines], page 426).

Emacs never reorders the text of a unibyte buffer, even if `bidi-display-reordering` is non-`nil` in the buffer. This is because unibyte buffers contain raw bytes, not characters, and thus lack the directionality properties required for reordering. Therefore, to test whether text in a buffer will be reordered for display, it is not enough to test the value of `bidi-display-reordering` alone. The correct test is this:

```
(if (and enable-multibyte-characters
         bidi-display-reordering)
    ;; Buffer is being reordered for display
  )
```

However, unibyte display and overlay strings *are* reordered if their parent buffer is reordered. This is because plain-ASCII strings are stored by Emacs as unibyte strings. If a unibyte display or overlay string includes non-ASCII characters, these characters are assumed to have left-to-right direction.

Text covered by `display` text properties, by overlays with `display` properties whose value is a string, and by any other properties that replace buffer text, is treated as a single unit when it is reordered for display. That is, the entire chunk of text covered by these properties is reordered together. Moreover, the bidirectional properties of the characters in such a chunk of text are ignored, and Emacs reorders them as if they were replaced with a single character `U+FFFC`, known as the *Object Replacement Character*. This means that placing a display property over a portion of text may change the way that the surrounding text is reordered for display. To prevent this unexpected effect, always place such properties on text whose directionality is identical with text that surrounds it.

Each paragraph of bidirectional text has a *base direction*, either right-to-left or left-to-right. Left-to-right paragraphs are displayed beginning at the left margin of the window, and are truncated or continued when the text reaches the right margin. Right-to-left paragraphs are displayed beginning at the right margin, and are continued or truncated at the left margin.

By default, Emacs determines the base direction of each paragraph by looking at the text at its beginning. The precise method of determining the base direction is specified by the UBA; in a nutshell, the first character in a paragraph that has an explicit directionality determines the base direction of the paragraph. However, sometimes a buffer may need to force a certain base direction for its paragraphs. For example, buffers containing program source code should force all paragraphs to be displayed left-to-right. You can use following variable to do this:

`bidi-paragraph-direction` [Variable]

If the value of this buffer-local variable is the symbol `right-to-left` or `left-to-right`, all paragraphs in the buffer are assumed to have that specified direction. Any other value is equivalent to `nil` (the default), which means to determine the base direction of each paragraph from its contents.

Modes for program source code should set this to `left-to-right`. Prog mode does this by default, so modes derived from Prog mode do not need to set this explicitly (see Section 20.2.5 [Basic Major Modes], page 407).

`current-bidi-paragraph-direction` **&optional** *buffer* [Function]
This function returns the paragraph direction at point in the named *buffer*. The returned value is a symbol, either `left-to-right` or `right-to-left`. If *buffer* is omitted or `nil`, it defaults to the current buffer. If the buffer-local value of the variable `bidi-paragraph-direction` is non-`nil`, the returned value will be identical to that value; otherwise, the returned value reflects the paragraph direction determined dynamically by Emacs. For buffers whose value of `bidi-display-reordering` is `nil` as well as unibyte buffers, this function always returns `left-to-right`.

Bidirectional reordering can have surprising and unpleasant effects when two strings with bidirectional content are juxtaposed in a buffer, or otherwise programmatically concatenated into a string of text. A typical problematic case is when a buffer consists of sequences of text "fields" separated by whitespace or punctuation characters, like Buffer Menu mode or Rmail Summary Mode. Because the punctuation characters used as separators have *weak directionality*, they take on the directionality of surrounding text. As result, a numeric field that follows a field with bidirectional content can be displayed *to the left* of the preceding field, messing up the expected layout. There are several ways to avoid this problem:

- Append the special character `U+200E`, LEFT-TO-RIGHT MARK, or LRM, to the end of each field that may have bidirectional content, or prepend it to the beginning of the following field. The function `bidi-string-mark-left-to-right`, described below, comes in handy for this purpose. (In a right-to-left paragraph, use `U+200F`, RIGHT-TO-LEFT MARK, or RLM, instead.) This is one of the solutions recommended by the UBA.
- Include the tab character in the field separator. The tab character plays the role of *segment separator* in bidirectional reordering, causing the text on either side to be reordered separately.
- Separate fields with a `display` property or overlay with a property value of the form `(space . PROPS)` (see Section 11.15.2 [Specified Space], page 164). Emacs treats this display specification as a *paragraph separator*, and reorders the text on either side separately.

`bidi-string-mark-left-to-right` *string* [Function]
This function returns its argument *string*, possibly modified, such that the result can be safely concatenated with another string, or juxtaposed with another string in a buffer, without disrupting the relative layout of this string and the next one on display. If the string returned by this function is displayed as part of a left-to-right paragraph, it will always appear on display to the left of the text that follows it. The function works by examining the characters of its argument, and if any of those characters could cause reordering on display, the function appends the LRM character to the string. The appended LRM character is made invisible by giving it an `invisible` text property of `t` (see Section 11.6 [Invisible Text], page 121).

The reordering algorithm uses the bidirectional properties of the characters stored as their `bidi-class` property (see ⟨undefined⟩ [Character Properties], page ⟨undefined⟩). Lisp

programs can change these properties by calling the `put-char-code-property` function. However, doing this requires a thorough understanding of the UBA, and is therefore not recommended. Any changes to the bidirectional properties of a character have global effect: they affect all Emacs frames and windows.

Similarly, the `mirroring` property is used to display the appropriate mirrored character in the reordered text. Lisp programs can affect the mirrored display by changing this property. Again, any such changes affect all of Emacs display.

# 12 Searching and Replacement

Like other editors, Emacs has commands to search for occurrences of a string. Emacs also has commands to replace occurrences of a string with a different string. There are also commands that do the same thing, but search for patterns instead of fixed strings.

You can also search multiple files under the control of a tags table (see Section 25.3.6 [Tags Search], page 578) or through the Dired `A` command (see Section 27.7 [Operating on Files], page 593), or ask the `grep` program to do it (see Section 24.4 [Grep Searching], page 537).

## 12.1 Incremental Search

The principal search command in Emacs is *incremental*: it begins searching as soon as you type the first character of the search string. As you type in the search string, Emacs shows you where the string (as you have typed it so far) would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you plan to do next, you may or may not need to terminate the search explicitly with RET.

`C-s` Incremental search forward (`isearch-forward`).

`C-r` Incremental search backward (`isearch-backward`).

### 12.1.1 Basics of Incremental Search

`C-s` Begin incremental search (`isearch-forward`).

`C-r` Begin reverse incremental search (`isearch-backward`).

`C-s` (`isearch-forward`) starts a forward incremental search. It reads characters from the keyboard, and moves point just past the end of the next occurrence of those characters in the buffer.

For instance, if you type `C-s` and then `F`, that puts the cursor after the first '`F`' that occurs in the buffer after the starting point. Then if you then type `O`, the cursor moves to just after the first '`FO`'; the '`F`' in that '`FO`' might not be the first '`F`' previously found. After another `O`, the cursor moves to just after the first '`FOO`'.

At each step, Emacs highlights the *current match*—the buffer text that matches the search string—using the `isearch` face (see Section 11.12 [Faces], page 137). The current search string is also displayed in the echo area.

If you make a mistake typing the search string, type DEL. Each DEL cancels the last character of the search string.

When you are satisfied with the place you have reached, type RET. This stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing `C-a` exits the search and then moves to the beginning of the line. RET is necessary only if the next command you want to type is a printing character, DEL, RET, or another character that is special within searches (`C-q`, `C-w`, `C-r`, `C-s`, `C-y`, `M-y`, `M-r`, `M-c`, `M-e`, and some others described below).

As a special exception, entering RET when the search string is empty launches nonincremental search (see Section 12.2 [Nonincremental Search], page 203).

When you exit the incremental search, it adds the original value of point to the mark ring, without activating the mark; you can thus use `C-u C-SPC` to return to where you were before beginning the search. See Section 8.4 [Mark Ring], page 92. It only does this if the mark was not already active.

To search backwards, use `C-r` (`isearch-backward`) instead of `C-s` to start the search. A backward search finds matches that end before the starting point, just as a forward search finds matches that begin after it.

### 12.1.2 Repeating Incremental Search

Suppose you search forward for '`FOO`' and find a match, but not the one you expected to find: the '`FOO`' you were aiming for occurs later in the buffer. In this event, type another `C-s` to move to the next occurrence of the search string. You can repeat this any number of times. If you overshoot, you can cancel some `C-s` characters with DEL. Similarly, each `C-r` in a backward incremental search repeats the backward search.

If you pause for a little while during incremental search, Emacs highlights all the other possible matches for the search string that are present on the screen. This helps you anticipate where you can get to by typing `C-s` or `C-r` to repeat the search. The other matches are highlighted differently from the current match, using the customizable face `lazy-highlight` (see Section 11.12 [Faces], page 137). If you don't like this feature, you can disable it by setting `isearch-lazy-highlight` to `nil`.

After exiting a search, you can search for the same string again by typing just `C-s C-s`. The first `C-s` is the key that invokes incremental search, and the second `C-s` means "search again". Similarly, `C-r C-r` searches backward for the last search string. In determining the last search string, it doesn't matter whether the string was searched for with `C-s` or `C-r`.

If you are searching forward but you realize you were looking for something before the starting point, type `C-r` to switch to a backward search, leaving the search string unchanged. Similarly, `C-s` in a backward search switches to a forward search.

If a search is failing and you ask to repeat it by typing another `C-s`, it starts again from the beginning of the buffer. Repeating a failing reverse search with `C-r` starts again from the end. This is called *wrapping around*, and '`Wrapped`' appears in the search prompt once this has happened. If you keep on going past the original starting point of the search, it changes to '`Overwrapped`', which means that you are revisiting matches that you have already seen.

To reuse earlier search strings, use the *search ring*. The commands `M-p` and `M-n` move through the ring to pick a search string to reuse. These commands leave the selected search ring element in the minibuffer, where you can edit it.

To edit the current search string in the minibuffer without replacing it with items from the search ring, type `M-e`. Type `C-s` or `C-r` to finish editing the string and search for it.

### 12.1.3 Errors in Incremental Search

If your string is not found at all, the echo area says '`Failing I-Search`', and the cursor moves past the place where Emacs found as much of your string as it could. Thus, if you search for '`FOOT`', and there is no '`FOOT`', you might see the cursor after the '`FOO`' in '`FOOL`'. In the echo area, the part of the search string that failed to match is highlighted using the face `isearch-fail`.

At this point, there are several things you can do. If your string was mistyped, you can use DEL to erase some of it and correct it. If you like the place you have found, you can type RET to remain there. Or you can type `C-g`, which removes from the search string the characters that could not be found (the '`T`' in '`FOOT`'), leaving those that were found (the '`FOO`' in '`FOOT`'). A second `C-g` at that point cancels the search entirely, returning point to where it was when the search started.

The quit command, `C-g`, does special things during searches; just what it does depends on the status of the search. If the search has found what you specified and is waiting for input, `C-g` cancels the entire search, moving the cursor back to where you started the search. If `C-g` is typed when there are characters in the search string that have not been found—because Emacs is still searching for them, or because it has failed to find them—then the search string characters which have not been found are discarded from the search string. With them gone, the search is now successful and waiting for more input, so a second `C-g` will cancel the entire search.

### 12.1.4 Special Input for Incremental Search

Some of the characters you type during incremental search have special effects.

By default, incremental search performs *lax space matching*: each space, or sequence of spaces, matches any sequence of one or more spaces in the text. Hence, '`foo bar`' matches '`foo bar`', '`foo bar`', '`foo bar`', and so on (but not '`foobar`'). More precisely, Emacs matches each sequence of space characters in the search string to a regular expression specified by the variable `search-whitespace-regexp`. For example, set it to '`"[[:space:]\n]+"`' to make spaces match sequences of newlines as well as spaces. To toggle lax space matching, type `M-s SPC` (`isearch-toggle-lax-whitespace`). To disable this feature entirely, change `search-whitespace-regexp` to `nil`; then each space in the search string matches exactly one space

If the search string you entered contains only lower-case letters, the search is case-insensitive; as long as an upper-case letter exists in the search string, the search becomes case-sensitive. If you delete the upper-case character from the search string, it ceases to have this effect. See Section 12.9 [Search Case], page 211.

To search for a newline character, type `C-j`.

To search for other control characters, such as CONTROL-S, quote it by typing `C-q` first (see Section 4.1 [Inserting Text], page 58). To search for non-ASCII characters, you can either use `C-q` and enter its octal code, or use an input method (see Section 19.4 [Input Methods], page 378). If an input method is enabled in the current buffer when you start the search, you can use it in the search string also. While typing the search string, you can toggle the input method with the command `C-\` (`isearch-toggle-input-method`). You can also turn on a non-default input method with `C-^` (`isearch-toggle-specified-input-method`), which prompts for the name of the input method. When an input method is active during incremental search, the search prompt includes the input method mnemonic, like this:

```
I-search [im]:
```

where *im* is the mnemonic of the active input method. Any input method you enable during incremental search remains enabled in the current buffer afterwards.

Typing `M-%` in incremental search invokes `query-replace` or `query-replace-regexp` (depending on search mode) with the current search string used as the string to replace. See Section 12.10.4 [Query Replace], page 213.

Typing `M-TAB` in incremental search invokes `isearch-complete`, which attempts to complete the search string using the search ring as a list of completion alternatives. See Section 5.4 [Completion], page 70. In many operating systems, the `M-TAB` key sequence is captured by the window manager; you then need to rebind `isearch-complete` to another key sequence if you want to use it (see Section 33.3.5 [Rebinding], page 705).

When incremental search is active, you can type `C-h C-h` to access interactive help options, including a list of special key bindings. These key bindings are part of the keymap `isearch-mode-map` (see Section 33.3.1 [Keymaps], page 703).

### 12.1.5 Isearch Yanking

Within incremental search, `C-y` (`isearch-yank-kill`) appends the current kill to the search string. `M-y` (`isearch-yank-pop`), if called after `C-y`, replaces that appended text with an earlier kill, similar to the usual `M-y` (`yank-pop`) command (see Section 22.8.3 [Yanking], page 465). `Mouse-2` appends the current X selection (see Section 9.3.2 [Primary Selection], page 101).

`C-w` (`isearch-yank-word-or-char`) appends the next character or word at point to the search string. This is an easy way to search for another occurrence of the text at point. (The decision of whether to copy a character or a word is heuristic.)

Similarly, `M-s C-e` (`isearch-yank-line`) appends the rest of the current line to the search string. If point is already at the end of a line, it appends the next line.

If the search is currently case-insensitive, both `C-w` and `M-s C-e` convert the text they copy to lower case, so that the search remains case-insensitive.

`C-M-w` (`isearch-del-char`) deletes the last character from the search string, and `C-M-y` (`isearch-yank-char`) appends the character after point to the search string. An alternative method to add the character after point is to enter the minibuffer with `M-e` (see Section 12.1.2 [Repeat Isearch], page 200) and type `C-f` at the end of the search string in the minibuffer.

### 12.1.6 Scrolling During Incremental Search

Normally, scrolling commands exit incremental search. If you change the variable `isearch-allow-scroll` to a non-`nil` value, that enables the use of the scroll-bar, as well as keyboard scrolling commands like `C-v`, `M-v`, and `C-l` (see ⟨undefined⟩ [Scrolling], page ⟨undefined⟩). This applies only to calling these commands via their bound key sequences—typing `M-x` will still exit the search. You can give prefix arguments to these commands in the usual way. This feature won't let you scroll the current match out of visibility, however.

The `isearch-allow-scroll` feature also affects some other commands, such as `C-x 2` (`split-window-below`) and `C-x ^` (`enlarge-window`), which don't exactly scroll but do affect where the text appears on the screen. It applies to any command whose name has a non-`nil` `isearch-scroll` property. So you can control which commands are affected by changing these properties.

For example, to make `C-h l` usable within an incremental search in all future Emacs sessions, use `C-h c` to find what command it runs (see ⟨undefined⟩ [Key Help], page ⟨unde-

fined⟩), which is `view-lossage`. Then you can put the following line in your init file (see Section 33.4 [Init File], page 711):

```
(put 'view-lossage 'isearch-scroll t)
```

This feature can be applied to any command that doesn't permanently change point, the buffer contents, the match data, the current buffer, or the selected window and frame. The command must not itself attempt an incremental search.

### 12.1.7 Searching the Minibuffer

If you start an incremental search while the minibuffer is active, Emacs searches the contents of the minibuffer. Unlike searching an ordinary buffer, the search string is not shown in the echo area, because that is used to display the minibuffer.

If an incremental search fails in the minibuffer, it tries searching the minibuffer history. See Section 5.5 [Minibuffer History], page 74. You can visualize the minibuffer and its history as a series of "pages", with the earliest history element on the first page and the current minibuffer on the last page. A forward search, `C-s`, searches forward to later pages; a reverse search, `C-r`, searches backwards to earlier pages. Like in ordinary buffer search, a failing search can wrap around, going from the last page to the first page or vice versa.

When the current match is on a history element, that history element is pulled into the minibuffer. If you exit the incremental search normally (e.g., by typing RET), it remains in the minibuffer afterwards. Canceling the search, with `C-g`, restores the contents of the minibuffer when you began the search.

## 12.2 Nonincremental Search

Emacs also has conventional nonincremental search commands, which require you to type the entire search string before searching begins.

`C-s RET` *`string`* `RET`
: Search for *string*.

`C-r RET` *`string`* `RET`
: Search backward for *string*.

To start a nonincremental search, first type `C-s RET`. This enters the minibuffer to read the search string; terminate the string with RET, and then the search takes place. If the string is not found, the search command signals an error.

When you type `C-s RET`, the `C-s` invokes incremental search as usual. That command is specially programmed to invoke the command for nonincremental search, `search-forward`, if the string you specify is empty. (Such an empty argument would otherwise be useless.) `C-r RET` does likewise, invoking the command `search-backward`.

## 12.3 Word Search

A *word search* finds a sequence of words without regard to the type of punctuation between them. For instance, if you enter a search string that consists of two words separated by a single space, the search matches any sequence of those two words separated by one or more spaces, newlines, or other punctuation characters. This is particularly useful for searching text documents, because you don't have to worry whether the words you are looking for are separated by newlines or spaces.

`M-s w` If incremental search is active, toggle word search mode (`isearch-toggle-word`); otherwise, begin an incremental forward word search (`isearch-forward-word`).

`M-s w RET` *words* `RET`
Search for *words*, using a forward nonincremental word search.

`M-s w C-r RET` *words* `RET`
Search backward for *words*, using a nonincremental word search.

To begin a forward incremental word search, type `M-s w`. If incremental search is not already active, this runs the command `isearch-forward-word`. If incremental search is already active (whether a forward or backward search), `M-s w` switches to a word search while keeping the direction of the search and the current search string unchanged. You can toggle word search back off by typing `M-s w` again.

To begin a nonincremental word search, type `M-s w RET` for a forward search, or `M-s w C-r RET` for a backward search. These run the commands `word-search-forward` and `word-search-backward` respectively.

Incremental and nonincremental word searches differ slightly in the way they find a match. In a nonincremental word search, the last word in the search string must exactly match a whole word. In an incremental word search, the matching is more lax: the last word in the search string can match part of a word, so that the matching proceeds incrementally as you type. This additional laxity does not apply to the lazy highlight, which always matches whole words.

## 12.4 Symbol Search

A *symbol search* is much like an ordinary search, except that the boundaries of the search must match the boundaries of a symbol. The meaning of *symbol* in this context depends on the major mode, and usually refers to a source code token, such as a Lisp symbol in Emacs Lisp mode. For instance, if you perform an incremental symbol search for the Lisp symbol `forward-word`, it would not match `isearch-forward-word`. This feature is thus mainly useful for searching source code.

`M-s _` If incremental search is active, toggle symbol search mode (`isearch-toggle-symbol`); otherwise, begin an incremental forward symbol search (`isearch-forward-symbol`).

`M-s _ RET` *symbol* `RET`
Search forward for *symbol*, nonincrementally.

`M-s _ C-r RET` *symbol* `RET`
Search backward for *symbol*, nonincrementally.

To begin a forward incremental symbol search, type `M-s _`. If incremental search is not already active, this runs the command `isearch-forward-symbol`. If incremental search is already active, `M-s _` switches to a symbol search, preserving the direction of the search and the current search string; you can disable symbol search by typing `M-s _` again. In incremental symbol search, only the beginning of the search string is required to match the beginning of a symbol.

To begin a nonincremental symbol search, type `M-s _ RET` for a forward search, or `M-s _ C-r RET` or a backward search. In nonincremental symbol searches, the beginning and end of the search string are required to match the beginning and end of a symbol, respectively.

## 12.5 Regular Expression Search

A *regular expression* (or *regexp* for short) is a pattern that denotes a class of alternative strings to match. Emacs provides both incremental and nonincremental ways to search for a match for a regexp. The syntax of regular expressions is explained in the next section.

`C-M-s` Begin incremental regexp search (`isearch-forward-regexp`).

`C-M-r` Begin reverse incremental regexp search (`isearch-backward-regexp`).

Incremental search for a regexp is done by typing `C-M-s` (`isearch-forward-regexp`), by invoking `C-s` with a prefix argument (whose value does not matter), or by typing `M-r` within a forward incremental search. This command reads a search string incrementally just like `C-s`, but it treats the search string as a regexp rather than looking for an exact match against the text in the buffer. Each time you add text to the search string, you make the regexp longer, and the new regexp is searched for. To search backward for a regexp, use `C-M-r` (`isearch-backward-regexp`), `C-r` with a prefix argument, or `M-r` within a backward incremental search.

All of the special key sequences in an ordinary incremental search do similar things in an incremental regexp search. For instance, typing `C-s` immediately after starting the search retrieves the last incremental search regexp used and searches forward for it. Incremental regexp and non-regexp searches have independent defaults. They also have separate search rings, which you can access with `M-p` and `M-n`.

Just as in ordinary incremental search, any SPC typed in incremental regexp search matches any sequence of one or more whitespace characters. The variable `search-whitespace-regexp` specifies the regexp for the lax space matching, and `M-s SPC` (`isearch-toggle-lax-whitespace`) toggles the feature. See Section 12.1.4 [Special Isearch], page 201.

In some cases, adding characters to the regexp in an incremental regexp search can make the cursor move back and start again. For example, if you have searched for '`foo`' and you add '`\|bar`', the cursor backs up in case the first '`bar`' precedes the first '`foo`'. See Section 12.6 [Regexps], page 206.

Forward and backward regexp search are not symmetrical, because regexp matching in Emacs always operates forward, starting with the beginning of the regexp. Thus, forward regexp search scans forward, trying a forward match at each possible starting position. Backward regexp search scans backward, trying a forward match at each possible starting position. These search methods are not mirror images.

Nonincremental search for a regexp is done with the commands `re-search-forward` and `re-search-backward`. You can invoke these with `M-x`, or by way of incremental regexp search with `C-M-s RET` and `C-M-r RET`.

If you use the incremental regexp search commands with a prefix argument, they perform ordinary string search, like `isearch-forward` and `isearch-backward`. See Section 12.1 [Incremental Search], page 199.

## 12.6 Syntax of Regular Expressions

This manual describes regular expression features that users typically use. See Section "Regular Expressions" in *The Emacs Lisp Reference Manual*, for additional features used mainly in Lisp programs.

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character matches that same character and nothing else. The special characters are '`$^.*+?[\`'. The character '`]`' is special if it ends a character alternative (see later). The character '`-`' is special inside a character alternative. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it. (When you use regular expressions in a Lisp program, each '`\`' must be doubled, see the example near the end of this section.)

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'. (When case distinctions are being ignored, these regexps also match '`F`' and '`O`', but we consider this a generalization of "the same string", rather than an exception.)

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression which matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string. For example, concatenating the regular expressions '`f`' and '`o`' gives the regular expression '`fo`', which matches only the string '`fo`'. Still trivial. To do something nontrivial, you need to use one of the special characters. Here is a list of them.

`.` (Period) is a special character that matches any single character except a newline. For example, the regular expressions '`a.b`' matches any three-character string that begins with '`a`' and ends with '`b`'.

`*` is not a construct by itself; it is a postfix operator that means to match the preceding regular expression repetitively any number of times, as many times as possible. Thus, '`o*`' matches any number of '`o`'s, including no '`o`'s.

'`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'. It matches '`f`', '`fo`', '`foo`', and so on.

The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`'-modified construct in case that makes it possible to match the rest of the pattern. For example, in matching '`ca*ar`' against the string '`caaar`', the '`a*`' first tries to match all three '`a`'s; but the rest of the pattern is '`ar`' and there is only '`r`' left to match, so this try fails. The next alternative is for '`a*`' to match only two '`a`'s. With this choice, the rest of the regexp matches successfully.

`+` is a postfix operator, similar to '`*`' except that it must match the preceding expression at least once. Thus, '`ca+r`' matches the strings '`car`' and '`caaaar`' but not the string '`cr`', whereas '`ca*r`' matches all three strings.

`?` is a postfix operator, similar to '`*`' except that it can match the preceding expression either once or not at all. Thus, '`ca?r`' matches '`car`' or '`cr`', and nothing else.

`*?`, `+?`, `??` are non-*greedy* variants of the operators above. The normal operators '`*`', '`+`', '`?`' match as much as they can, as long as the overall regexp can still match. With a following '`?`', they will match as little as possible.

Thus, both '`ab*`' and '`ab*?`' can match the string '`a`' and the string '`abbbb`'; but if you try to match them both against the text '`abbb`', '`ab*`' will match it all (the longest valid match), while '`ab*?`' will match just '`a`' (the shortest valid match).

Non-greedy operators match the shortest possible string starting at a given starting point; in a forward search, though, the earliest possible starting point for match is always the one chosen. Thus, if you search for '`a.*?$`' against the text '`abbab`' followed by a newline, it matches the whole string. Since it *can* match starting at the first '`a`', it does.

`\{`*n*`\}` is a postfix operator specifying *n* repetitions—that is, the preceding regular expression must match exactly *n* times in a row. For example, '`x\{4\}`' matches the string '`xxxx`' and nothing else.

`\{`*n*`,`*m*`\}` is a postfix operator specifying between *n* and *m* repetitions—that is, the preceding regular expression must match at least *n* times, but no more than *m* times. If *m* is omitted, then there is no upper limit, but the preceding regular expression must match at least *n* times.
'`\{0,1\}`' is equivalent to '`?`'.
'`\{0,\}`' is equivalent to '`*`'.
'`\{1,\}`' is equivalent to '`+`'.

`[ ... ]` is a *character set*, beginning with '`[`' and terminated by '`]`'.

In the simplest case, the characters between the two brackets are what this set can match. Thus, '`[ad]`' matches either one '`a`' or one '`d`', and '`[ad]*`' matches any string composed of just '`a`'s and '`d`'s (including the empty string). It follows that '`c[ad]*r`' matches '`cr`', '`car`', '`cdr`', '`caddaar`', etc.

You can also include character ranges in a character set, by writing the starting and ending characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case ASCII letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower-case ASCII letter or '`$`', '`%`' or period.

You can also include certain special *character classes* in a character set. A '`[:`' and balancing '`:]`' enclose a character class inside a character alternative. For instance, '`[[:alnum:]]`' matches any letter or digit. See Section "Char Classes" in *The Emacs Lisp Reference Manual*, for a list of character classes.

To include a '`]`' in a character set, you must make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', write '`-`' as the first or last character of the set, or put it after a range. Thus, '`[]-]`' matches both '`]`' and '`-`'.

To include '`^`' in a set, put it anywhere but at the beginning of the set. (At the beginning, it complements the set—see below.)

When you use a range in case-insensitive search, you should write both ends of the range in upper case, or both in lower case, or both should be non-letters.

The behavior of a mixed-case range such as ‘`A-z`’ is somewhat ill-defined, and it may change in future Emacs versions.

`[^ ... ]` ‘`[^`’ begins a *complemented character set*, which matches any character except the ones specified. Thus, ‘`[^a-z0-9A-Z]`’ matches all characters *except* ASCII letters and digits.

‘`^`’ is not special in a character set unless it is the first character. The character following the ‘`^`’ is treated as if it were first (in other words, ‘`-`’ and ‘`]`’ are not special there).

A complemented character set can match a newline, unless newline is mentioned as one of the characters not to match. This is in contrast to the handling of regexps in programs such as `grep`.

`^` is a special character that matches the empty string, but only at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, ‘`^foo`’ matches a ‘`foo`’ that occurs at the beginning of a line.

For historical compatibility reasons, ‘`^`’ can be used with this meaning only at the beginning of the regular expression, or after ‘`\(`’ or ‘`\|`’.

`$` is similar to ‘`^`’ but matches only at the end of a line. Thus, ‘`x+$`’ matches a string of one ‘`x`’ or more at the end of a line.

For historical compatibility reasons, ‘`$`’ can be used with this meaning only at the end of the regular expression, or before ‘`\)`’ or ‘`\|`’.

`\` has two functions: it quotes the special characters (including ‘`\`’), and it introduces additional special constructs.

Because ‘`\`’ quotes special characters, ‘`\$`’ is a regular expression that matches only ‘`$`’, and ‘`\[`’ is a regular expression that matches only ‘`[`’, and so on.

See the following section for the special constructs that begin with ‘`\`’.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, ‘`*foo`’ treats ‘`*`’ as ordinary since there is no preceding expression on which the ‘`*`’ can act. It is poor practice to depend on this behavior; it is better to quote the special character anyway, regardless of where it appears.

As a ‘`\`’ is not special inside a character alternative, it can never remove the special meaning of ‘`-`’ or ‘`]`’. So you should not quote these characters when they have no special meaning either. This would not clarify anything, since backslashes can legitimately precede these characters where they *have* special meaning, as in ‘`[^\]`’ (`"[^\\]"` for Lisp string syntax), which matches any single character except a backslash.

## 12.7 Backslash in Regular Expressions

For the most part, ‘`\`’ followed by any character matches only that character. However, there are several exceptions: two-character sequences starting with ‘`\`’ that have special meanings. The second character in the sequence is always an ordinary character when used on its own. Here is a table of ‘`\`’ constructs.

`\|` specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches some text if either *a* matches it or *b* matches it. It works by trying to match *a*, and if that fails, by trying to match *b*.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

'`\|`' applies to the largest possible surrounding expressions. Only a surrounding '`\( ... \)`' grouping can limit the grouping power of '`\|`'.

Full backtracking capability exists to handle multiple uses of '`\|`'.

`\( ... \)` is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix operators '`*`', '`+`' and '`?`' to operate on. Thus, '`ba\(na\)*`' matches '`bananana`', etc., with any (zero or more) number of '`na`' strings.
3. To record a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature that is assigned as a second meaning to the same '`\( ... \)`' construct. In practice there is usually no conflict between the two meanings; when there is a conflict, you can use a "shy" group.

`\(?: ... \)`
specifies a "shy" group that does not record the matched substring; you can't refer back to it with '`\d`'. This is useful in mechanically combining regular expressions, so that you can add groups for syntactic purposes without interfering with the numbering of the groups that are meant to be referred to.

`\d` matches the same text that matched the *d*th occurrence of a '`\( ... \)`' construct. This is called a *back reference*.

After the end of a '`\( ... \)`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`\`' followed by the digit *d* to mean "match the same text matched the *d*th time by the '`\( ... \)`' construct".

The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in the order that the open-parentheses appear in the regular expression. So you can use '`\1`' through '`\9`' to refer to the text matched by the corresponding '`\( ... \)`' constructs.

For example, '`\(.*\)\1`' matches any newline-free string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

If a particular '`\( ... \)`' construct matches more than once (which can easily happen if it is followed by '`*`'), only the last match is recorded.

`\‘` matches the empty string, but only at the beginning of the string or buffer (or its accessible portion) being matched against.

`\’` matches the empty string, but only at the end of the string or buffer (or its accessible portion) being matched against.

`\=` matches the empty string, but only at point.

`\b` matches the empty string, but only at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as a separate word. '`\bballs?\b`' matches '`ball`' or '`balls`' as a separate word.

'`\b`' matches at the beginning or end of the buffer regardless of what text appears next to it.

`\B` matches the empty string, but *not* at the beginning or end of a word.

`\<` matches the empty string, but only at the beginning of a word. '`\<`' matches at the beginning of the buffer only if a word-constituent character follows.

`\>` matches the empty string, but only at the end of a word. '`\>`' matches at the end of the buffer only if the contents end with a word-constituent character.

`\w` matches any word-constituent character. The syntax table determines which characters these are. See Section "Syntax Tables" in *The Emacs Lisp Reference Manual*.

`\W` matches any character that is not a word-constituent.

`\_<` matches the empty string, but only at the beginning of a symbol. A symbol is a sequence of one or more symbol-constituent characters. A symbol-constituent character is a character whose syntax is either '`w`' or '`_`'. '`\_<`' matches at the beginning of the buffer only if a symbol-constituent character follows.

`\_>` matches the empty string, but only at the end of a symbol. '`\_>`' matches at the end of the buffer only if the contents end with a symbol-constituent character.

`\sc` matches any character whose syntax is *c*. Here *c* is a character that designates a particular syntax class: thus, '`w`' for word constituent, '`-`' or ' ' for whitespace, '`.`' for ordinary punctuation, etc. See Section "Syntax Tables" in *The Emacs Lisp Reference Manual*.

`\Sc` matches any character whose syntax is not *c*.

`\cc` matches any character that belongs to the category *c*. For example, '`\cc`' matches Chinese characters, '`\cg`' matches Greek characters, etc. For the description of the known categories, type `M-x describe-categories RET`.

`\Cc` matches any character that does *not* belong to category *c*.

The constructs that pertain to words and syntax are controlled by the setting of the syntax table. See Section "Syntax Tables" in *The Emacs Lisp Reference Manual*.

## 12.8 Regular Expression Example

Here is an example of a regexp—similar to the regexp that Emacs uses, by default, to recognize the end of a sentence, not including the following space (i.e., the variable `sentence-end-base`):

```
[.?!][]\"')}]*
```

This contains two parts in succession: a character set matching period, '`?`', or '`!`', and a character set matching close-brackets, quotes, or parentheses, repeated zero or more times.

## 12.9 Searching and Case

Searches in Emacs normally ignore the case of the text they are searching through, if you specify the text in lower case. Thus, if you specify searching for 'foo', then 'Foo' and 'foo' also match. Regexps, and in particular character sets, behave likewise: '[ab]' matches 'a' or 'A' or 'b' or 'B'.

An upper-case letter anywhere in the incremental search string makes the search case-sensitive. Thus, searching for 'Foo' does not find 'foo' or 'FOO'. This applies to regular expression search as well as to string search. The effect ceases if you delete the upper-case letter from the search string.

Typing M-c within an incremental search toggles the case sensitivity of that search. The effect does not extend beyond the current incremental search to the next one, but it does override the effect of adding or removing an upper-case letter in the current search.

If you set the variable case-fold-search to nil, then all letters must match exactly, including case. This is a per-buffer variable; altering the variable normally affects only the current buffer, unless you change its default value. See Section 33.2.3 [Locals], page 697. This variable applies to nonincremental searches also, including those performed by the replace commands (see Section 12.10 [Replace], page 211) and the minibuffer history matching commands (see Section 5.5 [Minibuffer History], page 74).

Several related variables control case-sensitivity of searching and matching for specific commands or activities. For instance, tags-case-fold-search controls case sensitivity for find-tag. To find these variables, do M-x apropos-variable RET case-fold-search RET.

## 12.10 Replacement Commands

Emacs provides several commands for performing search-and-replace operations. In addition to the simple M-x replace-string command, there is M-% (query-replace), which presents each occurrence of the pattern and asks you whether to replace it.

The replace commands normally operate on the text from point to the end of the buffer. When the region is active, they operate on it instead (see Chapter 8 [Mark], page 89). The basic replace commands replace one *search string* (or regexp) with one *replacement string*. It is possible to perform several replacements in parallel, using the command expand-region-abbrevs (see ⟨undefined⟩ [Expanding Abbrevs], page ⟨undefined⟩).

Unlike incremental search, the replacement commands do not use lax space matching (see Section 12.1.4 [Special Isearch], page 201) by default. To enable lax space matching for replacement, change the variable replace-lax-whitespace to t. (This only affects how Emacs finds the text to replace, not the replacement text.)

### 12.10.1 Unconditional Replacement

M-x replace-string RET *string* RET *newstring* RET
: Replace every occurrence of *string* with *newstring*.

To replace every instance of 'foo' after point with 'bar', use the command M-x replace-string with the two arguments 'foo' and 'bar'. Replacement happens only in the text after point, so if you want to cover the whole buffer you must go to the beginning first. All occurrences up to the end of the buffer are replaced; to limit replacement to part

of the buffer, activate the region around that part. When the region is active, replacement is limited to the region (see Chapter 8 [Mark], page 89).

When `replace-string` exits, it leaves point at the last occurrence replaced. It adds the prior position of point (where the `replace-string` command was issued) to the mark ring, without activating the mark; use `C-u C-SPC` to move back there. See Section 8.4 [Mark Ring], page 92.

A prefix argument restricts replacement to matches that are surrounded by word boundaries.

See Section 12.10.3 [Replacement and Case], page 213, for details about case-sensitivity in replace commands.

### 12.10.2 Regexp Replacement

The `M-x replace-string` command replaces exact matches for a single string. The similar command `M-x replace-regexp` replaces any match for a specified pattern.

`M-x replace-regexp RET` *regexp* `RET` *newstring* `RET`
: Replace every match for *regexp* with *newstring*.

In `replace-regexp`, the *newstring* need not be constant: it can refer to all or part of what is matched by the *regexp*. '`\&`' in *newstring* stands for the entire match being replaced. '`\`*d*' in *newstring*, where *d* is a digit, stands for whatever matched the *d*th parenthesized grouping in *regexp*. (This is called a "back reference".) '`\#`' refers to the count of replacements already made in this command, as a decimal number. In the first replacement, '`\#`' stands for '`0`'; in the second, for '`1`'; and so on. For example,

```
M-x replace-regexp RET c[ad]+r RET \&-safe RET
```

replaces (for example) '`cadr`' with '`cadr-safe`' and '`cddr`' with '`cddr-safe`'.

```
M-x replace-regexp RET \(c[ad]+r\)-safe RET \1 RET
```

performs the inverse transformation. To include a '`\`' in the text to replace with, you must enter '`\\`'.

If you want to enter part of the replacement string by hand each time, use '`\?`' in the replacement string. Each replacement will ask you to edit the replacement string in the minibuffer, putting point where the '`\?`' was.

The remainder of this subsection is intended for specialized tasks and requires knowledge of Lisp. Most readers can skip it.

You can use Lisp expressions to calculate parts of the replacement string. To do this, write '`\,`' followed by the expression in the replacement string. Each replacement calculates the value of the expression and converts it to text without quoting (if it's a string, this means using the string's contents), and uses it in the replacement string in place of the expression itself. If the expression is a symbol, one space in the replacement string after the symbol name goes with the symbol name, so the value replaces them both.

Inside such an expression, you can use some special sequences. '`\&`' and '`\`*n*' refer here, as usual, to the entire match as a string, and to a submatch as a string. *n* may be multiple digits, and the value of '`\`*n*' is `nil` if subexpression *n* did not match. You can also use '`\#&`' and '`\#`*n*' to refer to those matches as numbers (this is valid when the match or submatch has the form of a numeral). '`\#`' here too stands for the number of already-completed replacements.

Repeating our example to exchange ‘x’ and ‘y’, we can thus do it also this way:

```
M-x replace-regexp RET \(x\)\|y RET
\,(if \1 "y" "x") RET
```

For computing replacement strings for ‘\,’, the `format` function is often useful (see Section “Formatting Strings” in *The Emacs Lisp Reference Manual*). For example, to add consecutively numbered strings like ‘ABC00042’ to columns 73 to 80 (unless they are already occupied), you can use

```
M-x replace-regexp RET ^.\{0,72\}$ RET
\,(format "%-72sABC%05d" \& \#) RET
```

### 12.10.3 Replace Commands and Case

If the first argument of a replace command is all lower case, the command ignores case while searching for occurrences to replace—provided `case-fold-search` is non-`nil`. If `case-fold-search` is set to `nil`, case is always significant in all searches.

In addition, when the *newstring* argument is all or partly lower case, replacement commands try to preserve the case pattern of each occurrence. Thus, the command

```
M-x replace-string RET foo RET bar RET
```

replaces a lower case ‘`foo`’ with a lower case ‘`bar`’, an all-caps ‘`FOO`’ with ‘`BAR`’, and a capitalized ‘`Foo`’ with ‘`Bar`’. (These three alternatives—lower case, all caps, and capitalized, are the only ones that `replace-string` can distinguish.)

If upper-case letters are used in the replacement string, they remain upper case every time that text is inserted. If upper-case letters are used in the first argument, the second argument is always substituted exactly as given, with no case conversion. Likewise, if either `case-replace` or `case-fold-search` is set to `nil`, replacement is done without case conversion.

### 12.10.4 Query Replace

`M-%` *string* `RET` *newstring* `RET`
: Replace some occurrences of *string* with *newstring*.

`C-M-%` *regexp* `RET` *newstring* `RET`
: Replace some matches for *regexp* with *newstring*.

If you want to change only some of the occurrences of ‘`foo`’ to ‘`bar`’, not all of them, use `M-%` (`query-replace`). This command finds occurrences of ‘`foo`’ one by one, displays each occurrence and asks you whether to replace it. Aside from querying, `query-replace` works just like `replace-string` (see Section 12.10.1 [Unconditional Replace], page 211). In particular, it preserves case provided `case-replace` is non-`nil`, as it normally is (see Section 12.10.3 [Replacement and Case], page 213). A numeric argument means to consider only occurrences that are bounded by word-delimiter characters.

`C-M-%` performs regexp search and replace (`query-replace-regexp`). It works like `replace-regexp` except that it queries like `query-replace`.

These commands highlight the current match using the face `query-replace`. They highlight other matches using `lazy-highlight` just like incremental search (see Section 12.1 [Incremental Search], page 199). By default, `query-replace-regexp` will show the substituted replacement string for the current match in the minibuffer. If you want to keep

special sequences '`\&`' and '`\n`' unexpanded, customize `query-replace-show-replacement` variable.

The characters you can type when you are shown a match for the string or regexp are:

SPC — to replace the occurrence with *newstring*.

DEL — to skip to the next occurrence without replacing this one.

`,` (Comma)
: to replace this occurrence and display the result. You are then asked for another input character to say what to do next. Since the replacement has already been made, DEL and SPC are equivalent in this situation; both move to the next occurrence.

  You can type `C-r` at this point (see below) to alter the replaced text. You can also type `C-x u` to undo the replacement; this exits the `query-replace`, so if you want to do further replacement you must use `C-x ESC ESC RET` to restart (see Section 5.6 [Repetition], page 76).

RET — to exit without doing any more replacements.

`.` (Period) — to replace this occurrence and then exit without searching for more occurrences.

`!` — to replace all remaining occurrences without asking again.

`Y` (Upper-case)
: to replace all remaining occurrences in all remaining buffers in multi-buffer replacements (like the Dired 'Q' command which performs query replace on selected files). It answers this question and all subsequent questions in the series with `"yes"`, without further user interaction.

`N` (Upper-case)
: to skip to the next buffer in multi-buffer replacements without replacing remaining occurrences in the current buffer. It answers this question `"no"`, gives up on the questions for the current buffer, and continues to the next buffer in the sequence.

`^` — to go back to the position of the previous occurrence (or what used to be an occurrence), in case you changed it by mistake or want to reexamine it.

`C-r` — to enter a recursive editing level, in case the occurrence needs to be edited rather than just replaced with *newstring*. When you are done, exit the recursive editing level with `C-M-c` to proceed to the next occurrence. See Section 31.9 [Recursive Edit], page 675.

`C-w` — to delete the occurrence, and then enter a recursive editing level as in `C-r`. Use the recursive edit to insert text to replace the deleted occurrence of *string*. When done, exit the recursive editing level with `C-M-c` to proceed to the next occurrence.

`e` — to edit the replacement string in the minibuffer. When you exit the minibuffer by typing RET, the minibuffer contents replace the current occurrence of the pattern. They also become the new replacement string for any further occurrences.

`C-l` to redisplay the screen. Then you must type another character to specify what to do with this occurrence.

`C-h` to display a message summarizing these options. Then you must type another character to specify what to do with this occurrence.

Some other characters are aliases for the ones listed above: `y`, `n` and `q` are equivalent to SPC, DEL and RET.

Aside from this, any other character exits the `query-replace`, and is then reread as part of a key sequence. Thus, if you type `C-k`, it exits the `query-replace` and then kills to end of line.

To restart a `query-replace` once it is exited, use `C-x ESC ESC`, which repeats the `query-replace` because it used the minibuffer to read its arguments. See Section 5.6 [Repetition], page 76.

See Section 27.7 [Operating on Files], page 593, for the Dired `Q` command which performs query replace on selected files. See also Section 27.9 [Transforming File Names], page 596, for Dired commands to rename, copy, or link files by replacing regexp matches in file names.

## 12.11 Other Search-and-Loop Commands

Here are some other commands that find matches for a regular expression. They all ignore case in matching, if the pattern contains no upper-case letters and `case-fold-search` is non-`nil`. Aside from `occur` and its variants, all operate on the text from point to the end of the buffer, or on the region if it is active.

`M-x multi-isearch-buffers`
: Prompt for one or more buffer names, ending with RET; then, begin a multi-buffer incremental search in those buffers. (If the search fails in one buffer, the next `C-s` tries searching the next specified buffer, and so forth.) With a prefix argument, prompt for a regexp and begin a multi-buffer incremental search in buffers matching that regexp.

`M-x multi-isearch-buffers-regexp`
: This command is just like `multi-isearch-buffers`, except it performs an incremental regexp search.

`M-x occur` Prompt for a regexp, and display a list showing each line in the buffer that contains a match for it. To limit the search to part of the buffer, narrow to that part (see ⟨undefined⟩ [Narrowing], page ⟨undefined⟩). A numeric argument *n* specifies that *n* lines of context are to be displayed before and after each matching line.

In the '`*Occur*`' buffer, you can click on each entry, or move point there and type RET, to visit the corresponding position in the buffer that was searched. `o` and `C-o` display the match in another window; `C-o` does not select it. Alternatively, you can use the `C-x ‘` (`next-error`) command to visit the occurrences one by one (see Section 24.2 [Compilation Mode], page 535).

Typing `e` in the '`*Occur*`' buffer switches to Occur Edit mode, in which edits made to the entries are also applied to the text in the originating buffer. Type `C-c C-c` to return to Occur mode.

The command `M-x list-matching-lines` is a synonym for `M-x occur`.

`M-s o` Run `occur` using the search string of the last incremental string search. You can also run `M-s o` when an incremental search is active; this uses the current search string.

`M-x multi-occur`
This command is just like `occur`, except it is able to search through multiple buffers. It asks you to specify the buffer names one by one.

`M-x multi-occur-in-matching-buffers`
This command is similar to `multi-occur`, except the buffers to search are specified by a regular expression that matches visited file names. With a prefix argument, it uses the regular expression to match buffer names instead.

`M-x how-many`
Prompt for a regexp, and print the number of matches for it in the buffer after point. If the region is active, this operates on the region instead.

`M-x flush-lines`
Prompt for a regexp, and delete each line that contains a match for it, operating on the text after point. This command deletes the current line if it contains a match starting after point. If the region is active, it operates on the region instead; if a line partially contained in the region contains a match entirely contained in the region, it is deleted.

If a match is split across lines, `flush-lines` deletes all those lines. It deletes the lines before starting to look for the next match; hence, it ignores a match starting on the same line at which another match ended.

`M-x keep-lines`
Prompt for a regexp, and delete each line that *does not* contain a match for it, operating on the text after point. If point is not at the beginning of a line, this command always keeps the current line. If the region is active, the command operates on the region instead; it never deletes lines that are only partially contained in the region (a newline that ends a line counts as part of that line).

If a match is split across lines, this command keeps all those lines.

# 13 Commands for Fixing Typos

In this chapter we describe commands that are useful when you catch a mistake while editing. The most fundamental of these commands is the undo command `C-/` (also bound to `C-x u` and `C-_`). This undoes a single command, or a part of a command (as in the case of `query-replace`), or several consecutive character insertions. Consecutive repetitions of `C-/` undo earlier and earlier changes, back to the limit of the undo information available.

Aside from the commands described here, you can erase text using deletion commands such as DEL (`delete-backward-char`). These were described earlier in this manual. See Section 4.3 [Erasing], page 61.

## 13.1 Undo

The *undo* command reverses recent changes in the buffer's text. Each buffer records changes individually, and the undo command always applies to the current buffer. You can undo all the changes in a buffer for as far back as the buffer's records go. Usually, each editing command makes a separate entry in the undo records, but some commands such as `query-replace` divide their changes into multiple entries for flexibility in undoing. Consecutive character insertion commands are usually grouped together into a single undo record, to make undoing less tedious.

`C-/`
`C-x u`
`C-_` Undo one entry in the current buffer's undo records (`undo`).

To begin to undo, type `C-/` (or its aliases, `C-_` or `C-x u`)[1]. This undoes the most recent change in the buffer, and moves point back to where it was before that change. Consecutive repetitions of `C-/` (or its aliases) undo earlier and earlier changes in the current buffer. If all the recorded changes have already been undone, the undo command signals an error.

Any command other than an undo command breaks the sequence of undo commands. Starting from that moment, the entire sequence of undo commands that you have just performed are themselves placed into the undo record, as a single set of changes. Therefore, to re-apply changes you have undone, type `C-f` or any other command that harmlessly breaks the sequence of undoing; then type `C-/` to undo the undo command.

Alternatively, if you want to resume undoing, without redoing previous undo commands, use `M-x undo-only`. This is like `undo`, but will not redo changes you have just undone.

If you notice that a buffer has been modified accidentally, the easiest way to recover is to type `C-/` repeatedly until the stars disappear from the front of the mode line (see Section 1.3 [Mode Line], page 8). Whenever an undo command makes the stars disappear from the mode line, it means that the buffer contents are the same as they were when the file was last read in or saved. If you do not remember whether you changed the buffer deliberately, type `C-/` once. When you see the last change you made undone, you will see whether it was an intentional change. If it was an accident, leave it undone. If it was deliberate, redo the change as described above.

[1] Aside from `C-/`, the `undo` command is also bound to `C-x u` because that is more straightforward for beginners to remember: 'u' stands for "undo". It is also bound to `C-_` because typing `C-/` on some text terminals actually enters `C-_`.

When there is an active region, any use of `undo` performs *selective undo*: it undoes the most recent change within the region, instead of the entire buffer. However, when Transient Mark mode is off (see Section 8.7 [Disabled Transient Mark], page 94), `C-/` always operates on the entire buffer, ignoring the region. In this case, you can perform selective undo by supplying a prefix argument to the `undo` command: `C-u C-/`. To undo further changes in the same region, repeat the `undo` command (no prefix argument is needed).

Some specialized buffers do not make undo records. Buffers whose names start with spaces never do; these buffers are used internally by Emacs to hold text that users don't normally look at or edit.

When the undo information for a buffer becomes too large, Emacs discards the oldest records from time to time (during *garbage collection*). You can specify how much undo information to keep by setting the variables `undo-limit`, `undo-strong-limit`, and `undo-outer-limit`. Their values are expressed in bytes.

The variable `undo-limit` sets a soft limit: Emacs keeps undo data for enough commands to reach this size, and perhaps exceed it, but does not keep data for any earlier commands beyond that. Its default value is 80000. The variable `undo-strong-limit` sets a stricter limit: any previous command (though not the most recent one) that pushes the size past this amount is forgotten. The default value of `undo-strong-limit` is 120000.

Regardless of the values of those variables, the most recent change is never discarded unless it gets bigger than `undo-outer-limit` (normally 12,000,000). At that point, Emacs discards the undo data and warns you about it. This is the only situation in which you cannot undo the last command. If this happens, you can increase the value of `undo-outer-limit` to make it even less likely to happen in the future. But if you didn't expect the command to create such large undo data, then it is probably a bug and you should report it. See Section 34.3 [Reporting Bugs], page 722.

## 13.2 Transposing Text

`C-t` Transpose two characters (`transpose-chars`).

`M-t` Transpose two words (`transpose-words`).

`C-M-t` Transpose two balanced expressions (`transpose-sexps`).

`C-x C-t` Transpose two lines (`transpose-lines`).

The common error of transposing two characters can be fixed, when they are adjacent, with the `C-t` command (`transpose-chars`). Normally, `C-t` transposes the two characters on either side of point. When given at the end of a line, rather than transposing the last character of the line with the newline, which would be useless, `C-t` transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a `C-t`. If you don't catch it so fast, you must move the cursor back between the two transposed characters before you type `C-t`. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (`C-r`) is often the best way. See Chapter 12 [Search], page 199.

`M-t` transposes the word before point with the word after point (`transpose-words`). It moves point forward over a word, dragging the word preceding or containing point forward as well. The punctuation characters between the words do not move. For example, '`FOO, BAR`' transposes into '`BAR, FOO`' rather than '`BAR FOO,`'.

`C-M-t` (`transpose-sexps`) is a similar command for transposing two expressions (see Section 23.4.1 [Expressions], page 521), and `C-x C-t` (`transpose-lines`) exchanges lines. They work like `M-t` except as regards what units of text they transpose.

A numeric argument to a transpose command serves as a repeat count: it tells the transpose command to move the character (word, expression, line) before or containing point across several other characters (words, expressions, lines). For example, `C-u 3 C-t` moves the character before point forward across three other characters. It would change '`f*oobar`' into '`oobf*ar`'. This is equivalent to repeating `C-t` three times. `C-u - 4 M-t` moves the word before point backward across four words. `C-u - C-M-t` would cancel the effect of plain `C-M-t`.

A numeric argument of zero is assigned a special meaning (because otherwise a command with a repeat count of zero would do nothing): to transpose the character (word, expression, line) ending after point with the one ending after the mark.

## 13.3 Case Conversion

`M-- M-l` Convert last word to lower case. Note `Meta--` is Meta-minus.

`M-- M-u` Convert last word to all upper case.

`M-- M-c` Convert last word to lower case with capital initial.

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands `M-l`, `M-u` and `M-c` have a special feature when used with a negative argument: they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See ⟨undefined⟩ [Case], page ⟨undefined⟩.

## 13.4 Checking and Correcting Spelling

This section describes the commands to check the spelling of a single word or of a portion of a buffer. These commands only work if the spelling checker program Aspell, Ispell or Hunspell is installed. These programs are not part of Emacs, but one of them is usually installed in GNU/Linux and other free operating systems.

`M-$` Check and correct spelling of the word at point (`ispell-word`). If the region is active, do it for all words in the region instead.

`M-x ispell`
Check and correct spelling of all words in the buffer. If the region is active, do it for all words in the region instead.

`M-x ispell-buffer`
Check and correct spelling in the buffer.

`M-x ispell-region`
Check and correct spelling in the region.

`M-x ispell-message`
Check and correct spelling in a draft mail message, excluding cited material.

`M-x ispell-change-dictionary RET` *dict* `RET`
Restart the Aspell/Ispell/Hunspell process, using *dict* as the dictionary.

`M-x ispell-kill-ispell`
: Kill the Aspell/Ispell/Hunspell subprocess.

`M-TAB`
`ESC TAB`
: Complete the word before point based on the spelling dictionary (`ispell-complete-word`).

`M-x flyspell-mode`
: Enable Flyspell mode, which highlights all misspelled words.

`M-x flyspell-prog-mode`
: Enable Flyspell mode for comments and strings only.

To check the spelling of the word around or before point, and optionally correct it as well, type `M-$` (`ispell-word`). If a region is active, `M-$` checks the spelling of all words within the region. See Chapter 8 [Mark], page 89. (When Transient Mark mode is off, `M-$` always acts on the word around or before point, ignoring the region; see Section 8.7 [Disabled Transient Mark], page 94.)

Similarly, the command `M-x ispell` performs spell-checking in the region if one is active, or in the entire buffer otherwise. The commands `M-x ispell-buffer` and `M-x ispell-region` explicitly perform spell-checking on the entire buffer or the region respectively. To check spelling in an email message you are writing, use `M-x ispell-message`; that command checks the whole buffer, except for material that is indented or appears to be cited from other messages. See Chapter 29 [Sending Mail], page 624.

When one of these commands encounters what appears to be an incorrect word, it asks you what to do. It usually displays a list of numbered "near-misses"—words that are close to the incorrect word. Then you must type a single-character response. Here are the valid responses:

*`digit`*
: Replace the word, just this time, with one of the displayed near-misses. Each near-miss is listed with a digit; type that digit to select it.

SPC
: Skip this word—continue to consider it incorrect, but don't change it here.

`r` *`new`* `RET`
: Replace the word, just this time, with *new*. (The replacement string will be rescanned for more spelling errors.)

`R` *`new`* `RET`
: Replace the word with *new*, and do a `query-replace` so you can replace it elsewhere in the buffer if you wish. (The replacements will be rescanned for more spelling errors.)

`a`
: Accept the incorrect word—treat it as correct, but only in this editing session.

`A`
: Accept the incorrect word—treat it as correct, but only in this editing session and for this buffer.

`i`
: Insert this word in your private dictionary file so that Aspell or Ispell or Hunspell will consider it correct from now on, even in future sessions.

`m`
: Like `i`, but you can also specify dictionary completion information.

`u`
: Insert the lower-case version of this word in your private dictionary file.

`l` *word* `RET`
: Look in the dictionary for words that match *word*. These words become the new list of "near-misses"; you can select one of them as the replacement by typing a digit. You can use '`*`' in *word* as a wildcard.

`C-g`

`X`
: Quit interactive spell checking, leaving point at the word that was being checked. You can restart checking again afterward with `C-u M-$`.

`x`
: Quit interactive spell checking and move point back to where it was when you started spell checking.

`q`
: Quit interactive spell checking and kill the spell-checker subprocess.

`?`
: Show the list of options.

In Text mode and related modes, `M-TAB` (`ispell-complete-word`) performs in-buffer completion based on spelling correction. Insert the beginning of a word, and then type `M-TAB`; this shows a list of completions. (If your window manager intercepts `M-TAB`, type `ESC TAB` or `C-M-i`.) Each completion is listed with a digit or character; type that digit or character to choose it.

Once started, the Aspell or Ispell or Hunspell subprocess continues to run, waiting for something to do, so that subsequent spell checking commands complete more quickly. If you want to get rid of the process, use `M-x ispell-kill-ispell`. This is not usually necessary, since the process uses no processor time except when you do spelling correction.

Ispell, Aspell and Hunspell look up spelling in two dictionaries: the standard dictionary and your personal dictionary. The standard dictionary is specified by the variable `ispell-local-dictionary` or, if that is `nil`, by the variable `ispell-dictionary`. If both are `nil`, the spelling program's default dictionary is used. The command `M-x ispell-change-dictionary` sets the standard dictionary for the buffer and then restarts the subprocess, so that it will use a different standard dictionary. Your personal dictionary is specified by the variable `ispell-personal-dictionary`. If that is `nil`, the spelling program looks for a personal dictionary in a default location.

A separate dictionary is used for word completion. The variable `ispell-complete-word-dict` specifies the file name of this dictionary. The completion dictionary must be different because it cannot use root and affix information. For some languages, there is a spell checking dictionary but no word completion dictionary.

Flyspell mode is a minor mode that performs automatic spell checking as you type. When it finds a word that it does not recognize, it highlights that word. Type `M-x flyspell-mode` to toggle Flyspell mode in the current buffer. To enable Flyspell mode in all text mode buffers, add `flyspell-mode` to `text-mode-hook`. See Section 33.2.2 [Hooks], page 696.

When Flyspell mode highlights a word as misspelled, you can click on it with `Mouse-2` to display a menu of possible corrections and actions. You can also correct the word by editing it manually in any way you like.

Flyspell Prog mode works just like ordinary Flyspell mode, except that it only checks words in comments and string constants. This feature is useful for editing programs. Type `M-x flyspell-prog-mode` to enable or disable this mode in the current buffer. To enable this mode in all programming mode buffers, add `flyspell-prog-mode` to `prog-mode-hook` (see Section 33.2.2 [Hooks], page 696).

# 14 Keyboard Macros

In this chapter we describe how to record a sequence of editing commands so you can repeat it conveniently later.

A *keyboard macro* is a command defined by an Emacs user to stand for another sequence of keys. For example, if you discover that you are about to type `C-n M-d C-d` forty times, you can speed your work by defining a keyboard macro to do `C-n M-d C-d`, and then executing it 39 more times.

You define a keyboard macro by executing and recording the commands which are its definition. Put differently, as you define a keyboard macro, the definition is being executed for the first time. This way, you can see the effects of your commands, so that you don't have to figure them out in your head. When you close the definition, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

Keyboard macros differ from ordinary Emacs commands in that they are written in the Emacs command language rather than in Lisp. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the Emacs command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, Lisp must be used.

## 14.1 Basic Use

F3 Start defining a keyboard macro (`kmacro-start-macro-or-insert-counter`).

F4 If a keyboard macro is being defined, end the definition; otherwise, execute the most recent keyboard macro (`kmacro-end-or-call-macro`).

`C-u F3` Re-execute last keyboard macro, then append keys to its definition.

`C-u C-u F3`
Append keys to the last keyboard macro without re-executing it.

`C-x C-k r` Run the last keyboard macro on each line that begins in the region (`apply-macro-to-region-lines`).

To start defining a keyboard macro, type F3. From then on, your keys continue to be executed, but also become part of the definition of the macro. '`Def`' appears in the mode line to remind you of what is going on. When you are finished, type F4 (`kmacro-end-or-call-macro`) to terminate the definition. For example,

```
F3 M-f foo F4
```

defines a macro to move forward a word and then insert '`foo`'. Note that F3 and F4 do not become part of the macro.

After defining the macro, you can call it with F4. For the above example, this has the same effect as typing `M-f foo` again. (Note the two roles of the F4 command: it ends the macro if you are in the process of defining one, or calls the last macro otherwise.) You can also supply F4 with a numeric prefix argument '`n`', which means to invoke the macro '`n`' times. An argument of zero repeats the macro indefinitely, until it gets an error or you type `C-g` (or, on MS-DOS, `C-BREAK`).

The above example demonstrates a handy trick that you can employ with keyboard macros: if you wish to repeat an operation at regularly spaced places in the text, include a motion command as part of the macro. In this case, repeating the macro inserts the string '`foo`' after each successive word.

After terminating the definition of a keyboard macro, you can append more keystrokes to its definition by typing `C-u F3`. This is equivalent to plain F3 followed by retyping the whole definition so far. As a consequence, it re-executes the macro as previously defined. If you change the variable `kmacro-execute-before-append` to `nil`, the existing macro will not be re-executed before appending to it (the default is `t`). You can also add to the end of the definition of the last keyboard macro without re-executing it by typing `C-u C-u F3`.

When a command reads an argument with the minibuffer, your minibuffer input becomes part of the macro along with the command. So when you replay the macro, the command gets the same argument as when you entered the macro. For example,

```
F3 C-a C-k C-x b foo RET C-y C-x b RET F4
```

defines a macro that kills the current line, yanks it into the buffer '`foo`', then returns to the original buffer.

Most keyboard commands work as usual in a keyboard macro definition, with some exceptions. Typing `C-g` (`keyboard-quit`) quits the keyboard macro definition. Typing `C-M-c` (`exit-recursive-edit`) can be unreliable: it works as you'd expect if exiting a recursive edit that started within the macro, but if it exits a recursive edit that started before you invoked the keyboard macro, it also necessarily exits the keyboard macro too. Mouse events are also unreliable, even though you can use them in a keyboard macro: when the macro replays the mouse event, it uses the original mouse position of that event, the position that the mouse had while you were defining the macro. The effect of this may be hard to predict.

The command `C-x C-k r` (`apply-macro-to-region-lines`) repeats the last defined keyboard macro on each line that begins in the region. It does this line by line, by moving point to the beginning of the line and then executing the macro.

In addition to the F3 and F4 commands described above, Emacs also supports an older set of key bindings for defining and executing keyboard macros. To begin a macro definition, type `C-x (` (`kmacro-start-macro`); as with F3, a prefix argument appends this definition to the last keyboard macro. To end a macro definition, type `C-x )` (`kmacro-end-macro`). To execute the most recent macro, type `C-x e` (`kmacro-end-and-call-macro`). If you enter `C-x e` while defining a macro, the macro is terminated and executed immediately. Immediately after typing `C-x e`, you can type E repeatedly to immediately repeat the macro one or more times. You can also give `C-x e` a repeat argument, just like F4.

`C-x )` can be given a repeat count as an argument. This means to repeat the macro right after defining it. The macro definition itself counts as the first repetition, since it is executed as you define it, so `C-u 4 C-x )` executes the macro immediately 3 additional times.

## 14.2 The Keyboard Macro Ring

All defined keyboard macros are recorded in the *keyboard macro ring*. There is only one keyboard macro ring, shared by all buffers.

`C-x C-k C-k`
: Execute the keyboard macro at the head of the ring (`kmacro-end-or-call-macro-repeat`).

`C-x C-k C-n`
: Rotate the keyboard macro ring to the next macro (defined earlier) (`kmacro-cycle-ring-next`).

`C-x C-k C-p`
: Rotate the keyboard macro ring to the previous macro (defined later) (`kmacro-cycle-ring-previous`).

All commands which operate on the keyboard macro ring use the same `C-x C-k` prefix. Most of these commands can be executed and repeated immediately after each other without repeating the `C-x C-k` prefix. For example,

```
C-x C-k C-p C-p C-k C-k C-k C-n C-n C-k C-p C-k C-d
```

will rotate the keyboard macro ring to the "second previous" macro, execute the resulting head macro three times, rotate back to the original head macro, execute that once, rotate to the "previous" macro, execute that, and finally delete it from the macro ring.

The command `C-x C-k C-k` (`kmacro-end-or-call-macro-repeat`) executes the keyboard macro at the head of the macro ring. You can repeat the macro immediately by typing another `C-k`, or you can rotate the macro ring immediately by typing `C-n` or `C-p`.

When a keyboard macro is being defined, `C-x C-k C-k` behaves like F4 except that, immediately afterward, you can use most key bindings of this section without the `C-x C-k` prefix. For instance, another `C-k` will re-execute the macro.

The commands `C-x C-k C-n` (`kmacro-cycle-ring-next`) and `C-x C-k C-p` (`kmacro-cycle-ring-previous`) rotate the macro ring, bringing the next or previous keyboard macro to the head of the macro ring. The definition of the new head macro is displayed in the echo area. You can continue to rotate the macro ring immediately by repeating just `C-n` and `C-p` until the desired macro is at the head of the ring. To execute the new macro ring head immediately, just type `C-k`.

Note that Emacs treats the head of the macro ring as the "last defined keyboard macro". For instance, F4 will execute that macro, and `C-x C-k n` will give it a name.

The maximum number of macros stored in the keyboard macro ring is determined by the customizable variable `kmacro-ring-max`.

## 14.3 The Keyboard Macro Counter

Each keyboard macro has an associated counter, which is initialized to 0 when you start defining the macro. This counter allows you to insert a number into the buffer that depends on the number of times the macro has been called. The counter is incremented each time its value is inserted into the buffer.

F3
: In a keyboard macro definition, insert the keyboard macro counter value in the buffer (`kmacro-start-macro-or-insert-counter`).

`C-x C-k C-i`
: Insert the keyboard macro counter value in the buffer (`kmacro-insert-counter`).

`C-x C-k C-c`
: Set the keyboard macro counter (`kmacro-set-counter`).

`C-x C-k C-a`
: Add the prefix arg to the keyboard macro counter (`kmacro-add-counter`).

`C-x C-k C-f`
: Specify the format for inserting the keyboard macro counter (`kmacro-set-format`).

When you are defining a keyboard macro, the command F3 (`kmacro-start-macro-or-insert-counter`) inserts the current value of the keyboard macro's counter into the buffer, and increments the counter by 1. (If you are not defining a macro, F3 begins a macro definition instead. See Section 14.1 [Basic Keyboard Macro], page 222.) You can use a numeric prefix argument to specify a different increment. If you just specify a `C-u` prefix, that is the same as an increment of zero: it inserts the current counter value without changing it.

As an example, let us show how the keyboard macro counter can be used to build a numbered list. Consider the following key sequence:

```
F3 C-a F3 . SPC F4
```

As part of this keyboard macro definition, the string '`0. `' was inserted into the beginning of the current line. If you now move somewhere else in the buffer and type F4 to invoke the macro, the string '`1. `' is inserted at the beginning of that line. Subsequent invocations insert '`2. `', '`3. `', and so forth.

The command `C-x C-k C-i` (`kmacro-insert-counter`) does the same thing as F3, but it can be used outside a keyboard macro definition. When no keyboard macro is being defined or executed, it inserts and increments the counter of the macro at the head of the keyboard macro ring.

The command `C-x C-k C-c` (`kmacro-set-counter`) sets the current macro counter to the value of the numeric argument. If you use it inside the macro, it operates on each repetition of the macro. If you specify just `C-u` as the prefix, while executing the macro, that resets the counter to the value it had at the beginning of the current repetition of the macro (undoing any increments so far in this repetition).

The command `C-x C-k C-a` (`kmacro-add-counter`) adds the prefix argument to the current macro counter. With just `C-u` as argument, it resets the counter to the last value inserted by any keyboard macro. (Normally, when you use this, the last insertion will be in the same macro and it will be the same counter.)

The command `C-x C-k C-f` (`kmacro-set-format`) prompts for the format to use when inserting the macro counter. The default format is '`%d`', which means to insert the number in decimal without any padding. You can exit with empty minibuffer to reset the format to this default. You can specify any format string that the `format` function accepts and that makes sense with a single integer extra argument (see Section "Formatting Strings" in *The Emacs Lisp Reference Manual*). Do not put the format string inside double quotes when you insert it in the minibuffer.

If you use this command while no keyboard macro is being defined or executed, the new format affects all subsequent macro definitions. Existing macros continue to use the format in effect when they were defined. If you set the format while defining a keyboard macro,

this affects the macro being defined from that point on, but it does not affect subsequent macros. Execution of the macro will, at each step, use the format in effect at that step during its definition. Changes to the macro format during execution of a macro, like the corresponding changes during its definition, have no effect on subsequent macros.

The format set by `C-x C-k C-f` does not affect insertion of numbers stored in registers.

If you use a register as a counter, incrementing it on each repetition of the macro, that accomplishes the same thing as a keyboard macro counter. See Section 10.5 [Number Registers], page 108. For most purposes, it is simpler to use a keyboard macro counter.

## 14.4 Executing Macros with Variations

In a keyboard macro, you can create an effect similar to that of `query-replace`, in that the macro asks you each time around whether to make a change.

`C-x q` When this point is reached during macro execution, ask for confirmation (`kbd-macro-query`).

While defining the macro, type `C-x q` at the point where you want the query to occur. During macro definition, the `C-x q` does nothing, but when you run the macro later, `C-x q` asks you interactively whether to continue.

The valid responses when `C-x q` asks are:

SPC (or `y`)
: Continue executing the keyboard macro.

DEL (or `n`)
: Skip the remainder of this repetition of the macro, and start right away with the next repetition.

RET (or `q`)
: Skip the remainder of this repetition and cancel further repetitions.

`C-r` Enter a recursive editing level, in which you can perform editing which is not part of the macro. When you exit the recursive edit using `C-M-c`, you are asked again how to continue with the keyboard macro. If you type a SPC at this time, the rest of the macro definition is executed. It is up to you to leave point and the text in a state such that the rest of the macro will do what you want.

`C-u C-x q`, which is `C-x q` with a numeric argument, performs a completely different function. It enters a recursive edit reading input from the keyboard, both when you type it during the definition of the macro, and when it is executed from the macro. During definition, the editing you do inside the recursive edit does not become part of the macro. During macro execution, the recursive edit gives you a chance to do some particularized editing on each repetition. See Section 31.9 [Recursive Edit], page 675.

## 14.5 Naming and Saving Keyboard Macros

`C-x C-k n` Give a command name (for the duration of the Emacs session) to the most recently defined keyboard macro (`kmacro-name-last-macro`).

`C-x C-k b` Bind the most recently defined keyboard macro to a key sequence (for the duration of the session) (`kmacro-bind-to-key`).

`M-x insert-kbd-macro`
: Insert in the buffer a keyboard macro's definition, as Lisp code.

If you wish to save a keyboard macro for later use, you can give it a name using `C-x C-k n` (`kmacro-name-last-macro`). This reads a name as an argument using the minibuffer and defines that name to execute the last keyboard macro, in its current form. (If you later add to the definition of this macro, that does not alter the name's definition as a macro.) The macro name is a Lisp symbol, and defining it in this way makes it a valid command name for calling with `M-x` or for binding a key to with `global-set-key` (see Section 33.3.1 [Keymaps], page 703). If you specify a name that has a prior definition other than a keyboard macro, an error message is shown and nothing is changed.

You can also bind the last keyboard macro (in its current form) to a key, using `C-x C-k b` (`kmacro-bind-to-key`) followed by the key sequence you want to bind. You can bind to any key sequence in the global keymap, but since most key sequences already have other bindings, you should select the key sequence carefully. If you try to bind to a key sequence with an existing binding (in any keymap), this command asks you for confirmation before replacing the existing binding.

To avoid problems caused by overriding existing bindings, the key sequences `C-x C-k 0` through `C-x C-k 9` and `C-x C-k A` through `C-x C-k Z` are reserved for your own keyboard macro bindings. In fact, to bind to one of these key sequences, you only need to type the digit or letter rather than the whole key sequences. For example,

```
C-x C-k b 4
```

will bind the last keyboard macro to the key sequence `C-x C-k 4`.

Once a macro has a command name, you can save its definition in a file. Then it can be used in another editing session. First, visit the file you want to save the definition in. Then use this command:

```
M-x insert-kbd-macro RET macroname RET
```

This inserts some Lisp code that, when executed later, will define the same macro with the same definition it has now. (You need not understand Lisp code to do this, because `insert-kbd-macro` writes the Lisp code for you.) Then save the file. You can load the file later with `load-file` (see Section 24.8 [Lisp Libraries], page 549). If the file you save in is your init file '`~/.emacs`' (see Section 33.4 [Init File], page 711) then the macro will be defined each time you run Emacs.

If you give `insert-kbd-macro` a numeric argument, it makes additional Lisp code to record the keys (if any) that you have bound to *macroname*, so that the macro will be reassigned the same keys when you load the file.

## 14.6 Editing a Keyboard Macro

`C-x C-k C-e`
: Edit the last defined keyboard macro (`kmacro-edit-macro`).

`C-x C-k e name RET`
: Edit a previously defined keyboard macro *name* (`edit-kbd-macro`).

`C-x C-k l` Edit the last 300 keystrokes as a keyboard macro (`kmacro-edit-lossage`).

You can edit the last keyboard macro by typing `C-x C-k C-e` or `C-x C-k RET` (`kmacro-edit-macro`). This formats the macro definition in a buffer and enters a specialized major mode for editing it. Type `C-h m` once in that buffer to display details of how to edit the macro. When you are finished editing, type `C-c C-c`.

You can edit a named keyboard macro or a macro bound to a key by typing `C-x C-k e` (`edit-kbd-macro`). Follow that with the keyboard input that you would use to invoke the macro—`C-x e` or `M-x` *name* or some other key sequence.

You can edit the last 300 keystrokes as a macro by typing `C-x C-k l` (`kmacro-edit-lossage`).

## 14.7 Stepwise Editing a Keyboard Macro

You can interactively replay and edit the last keyboard macro, one command at a time, by typing `C-x C-k SPC` (`kmacro-step-edit-macro`). Unless you quit the macro using `q` or `C-g`, the edited macro replaces the last macro on the macro ring.

This macro editing feature shows the last macro in the minibuffer together with the first (or next) command to be executed, and prompts you for an action. You can enter `?` to get a summary of your options. These actions are available:

- `SPC` and `y` execute the current command, and advance to the next command in the keyboard macro.
- `n`, `d`, and `DEL` skip and delete the current command.
- `f` skips the current command in this execution of the keyboard macro, but doesn't delete it from the macro.
- TAB executes the current command, as well as all similar commands immediately following the current command; for example, TAB may be used to insert a sequence of characters (corresponding to a sequence of `self-insert-command` commands).
- `c` continues execution (without further editing) until the end of the keyboard macro. If execution terminates normally, the edited macro replaces the original keyboard macro.
- `C-k` skips and deletes the rest of the keyboard macro, terminates step-editing, and replaces the original keyboard macro with the edited macro.
- `q` and `C-g` cancels the step-editing of the keyboard macro; discarding any changes made to the keyboard macro.
- `i KEY... C-j` reads and executes a series of key sequences (not including the final `C-j`), and inserts them before the current command in the keyboard macro, without advancing over the current command.
- `I KEY...` reads one key sequence, executes it, and inserts it before the current command in the keyboard macro, without advancing over the current command.
- `r KEY... C-j` reads and executes a series of key sequences (not including the final `C-j`), and replaces the current command in the keyboard macro with them, advancing over the inserted key sequences.
- `R KEY...` reads one key sequence, executes it, and replaces the current command in the keyboard macro with that key sequence, advancing over the inserted key sequence.
- `a KEY... C-j` executes the current command, then reads and executes a series of key sequences (not including the final `C-j`), and inserts them after the current command

in the keyboard macro; it then advances over the current command and the inserted key sequences.

- `A KEY... C-j` executes the rest of the commands in the keyboard macro, then reads and executes a series of key sequences (not including the final `C-j`), and appends them at the end of the keyboard macro; it then terminates the step-editing and replaces the original keyboard macro with the edited macro.

# 15 Files

This chapter describes the Emacs Lisp functions and variables to find, create, view, save, and otherwise work with files and file directories. A few other file-related functions are described in Chapter 16 [Buffers], page 272, and those related to backups and auto-saving are described in ⟨undefined⟩ [Backups and Auto-Saving], page ⟨undefined⟩.

Many of the file functions take one or more arguments that are file names. A file name is actually a string. Most of these functions expand file name arguments by calling `expand-file-name`, so that '`~`' is handled correctly, as are relative file names (including '`../`'). See Section 15.8.4 [File Name Expansion], page 255.

In addition, certain *magic* file names are handled specially. For example, when a remote file name is specified, Emacs accesses the file over the network via an appropriate protocol (see Section "Remote Files" in *The GNU Emacs Manual*). This handling is done at a very low level, so you may assume that all the functions described in this chapter accept magic file names as file name arguments, except where noted. See Section 15.11 [Magic File Names], page 262, for details.

When file I/O functions signal Lisp errors, they usually use the condition `file-error` (see ⟨undefined⟩ [Handling Errors], page ⟨undefined⟩). The error message is in most cases obtained from the operating system, according to locale `system-messages-locale`, and decoded using coding system `locale-coding-system` (see ⟨undefined⟩ [Locales], page ⟨undefined⟩).

## 15.1 Visiting Files

Visiting a file means reading a file into a buffer. Once this is done, we say that the buffer is *visiting* that file, and call the file "the visited file" of the buffer.

A file and a buffer are two different things. A file is information recorded permanently in the computer (unless you delete it). A buffer, on the other hand, is information inside of Emacs that will vanish at the end of the editing session (or when you kill the buffer). Usually, a buffer contains information that you have copied from a file; then we say the buffer is visiting that file. The copy in the buffer is what you modify with editing commands. Such changes to the buffer do not change the file; therefore, to make the changes permanent, you must *save* the buffer, which means copying the altered buffer contents back into the file.

In spite of the distinction between files and buffers, people often refer to a file when they mean a buffer and vice-versa. Indeed, we say, "I am editing a file", rather than, "I am editing a buffer that I will soon save as a file of the same name". Humans do not usually need to make the distinction explicit. When dealing with a computer program, however, it is good to keep the distinction in mind.

### 15.1.1 Functions for Visiting Files

This section describes the functions normally used to visit files. For historical reasons, these functions have names starting with '`find-`' rather than '`visit-`'. See Section 16.4 [Buffer File Name], page 276, for functions and variables that access the visited file name of a buffer or that find an existing buffer by its visited file name.

In a Lisp program, if you want to look at the contents of a file but not alter it, the fastest way is to use `insert-file-contents` in a temporary buffer. Visiting the file is not necessary and takes longer. See Section 15.3 [Reading from Files], page 236.

`find-file` *filename* **&optional** *wildcards* [Command]

This command selects a buffer visiting the file *filename*, using an existing buffer if there is one, and otherwise creating a new buffer and reading the file into it. It also returns that buffer.

Aside from some technical details, the body of the `find-file` function is basically equivalent to:

```
(switch-to-buffer (find-file-noselect filename nil nil wildcards))
```

(See `switch-to-buffer` in Section 17.11 [Switching Buffers], page 311.)

If *wildcards* is non-`nil`, which is always true in an interactive call, then `find-file` expands wildcard characters in *filename* and visits all the matching files.

When `find-file` is called interactively, it prompts for *filename* in the minibuffer.

`find-file-literally` *filename* [Command]

This command visits *filename*, like `find-file` does, but it does not perform any format conversions (see Section 15.12 [Format Conversion], page 267), character code conversions (see Section 19.6 [Coding Systems], page 381), or end-of-line conversions (see ⟨undefined⟩ [Coding System Basics], page ⟨undefined⟩). The buffer visiting the file is made unibyte, and its major mode is Fundamental mode, regardless of the file name. File local variable specifications in the file (see ⟨undefined⟩ [File Local Variables], page ⟨undefined⟩) are ignored, and automatic decompression and adding a newline at the end of the file due to `require-final-newline` (see Section 15.2 [Saving Buffers], page 234) are also disabled.

Note that if Emacs already has a buffer visiting the same file non-literally, it will not visit the same file literally, but instead just switch to the existing buffer. If you want to be sure of accessing a file's contents literally, you should create a temporary buffer and then read the file contents into it using `insert-file-contents-literally` (see Section 15.3 [Reading from Files], page 236).

`find-file-noselect` *filename* **&optional** *nowarn rawfile wildcards* [Function]

This function is the guts of all the file-visiting functions. It returns a buffer visiting the file *filename*. You may make the buffer current or display it in a window if you wish, but this function does not do so.

The function returns an existing buffer if there is one; otherwise it creates a new buffer and reads the file into it. When `find-file-noselect` uses an existing buffer, it first verifies that the file has not changed since it was last visited or saved in that buffer. If the file has changed, this function asks the user whether to reread the changed file. If the user says '`yes`', any edits previously made in the buffer are lost.

Reading the file involves decoding the file's contents (see Section 19.6 [Coding Systems], page 381), including end-of-line conversion, and format conversion (see Section 15.12 [Format Conversion], page 267). If *wildcards* is non-`nil`, then `find-file-noselect` expands wildcard characters in *filename* and visits all the matching files.

This function displays warning or advisory messages in various peculiar cases, unless the optional argument *nowarn* is non-`nil`. For example, if it needs to create a buffer, and there is no file named *filename*, it displays the message '`(New file)`' in the echo area, and leaves the buffer empty.

The `find-file-noselect` function normally calls `after-find-file` after reading the file (see Section 15.1.2 [Subroutines of Visiting], page 233). That function sets the buffer major mode, parses local variables, warns the user if there exists an auto-save file more recent than the file just visited, and finishes by running the functions in `find-file-hook`.

If the optional argument *rawfile* is non-`nil`, then `after-find-file` is not called, and the `find-file-not-found-functions` are not run in case of failure. What's more, a non-`nil` *rawfile* value suppresses coding system conversion and format conversion.

The `find-file-noselect` function usually returns the buffer that is visiting the file *filename*. But, if wildcards are actually used and expanded, it returns a list of buffers that are visiting the various files.

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

`find-file-other-window` *filename* **&optional** *wildcards* [Command]
This command selects a buffer visiting the file *filename*, but does so in a window other than the selected window. It may use another existing window or split a window; see Section 17.11 [Switching Buffers], page 311.

When this command is called interactively, it prompts for *filename*.

`find-file-read-only` *filename* **&optional** *wildcards* [Command]
This command selects a buffer visiting the file *filename*, like `find-file`, but it marks the buffer as read-only. See Section 16.7 [Read Only Buffers], page 280, for related functions and variables.

When this command is called interactively, it prompts for *filename*.

`find-file-wildcards` [User Option]
If this variable is non-`nil`, then the various `find-file` commands check for wildcard characters and visit all the files that match them (when invoked interactively or when their *wildcards* argument is non-`nil`). If this option is `nil`, then the `find-file` commands ignore their *wildcards* argument and never treat wildcard characters specially.

`find-file-hook` [User Option]
The value of this variable is a list of functions to be called after a file is visited. The file's local-variables specification (if any) will have been processed before the hooks are run. The buffer visiting the file is current when the hook functions are run.

This variable is a normal hook. See Section 33.2.2 [Hooks], page 696.

`find-file-not-found-functions` [Variable]
The value of this variable is a list of functions to be called when `find-file` or `find-file-noselect` is passed a nonexistent file name. `find-file-noselect` calls these

functions as soon as it detects a nonexistent file. It calls them in the order of the list, until one of them returns non-`nil`. `buffer-file-name` is already set up.

This is not a normal hook because the values of the functions are used, and in many cases only some of the functions are called.

`find-file-literally` [Variable]

This buffer-local variable, if set to a non-`nil` value, makes `save-buffer` behave as if the buffer were visiting its file literally, i.e., without conversions of any kind. The command `find-file-literally` sets this variable's local value, but other equivalent functions and commands can do that as well, e.g., to avoid automatic addition of a newline at the end of the file. This variable is permanent local, so it is unaffected by changes of major modes.

### 15.1.2 Subroutines of Visiting

The `find-file-noselect` function uses two important subroutines which are sometimes useful in user Lisp code: `create-file-buffer` and `after-find-file`. This section explains how to use them.

`create-file-buffer` *filename* [Function]

This function creates a suitably named buffer for visiting *filename*, and returns it. It uses *filename* (sans directory) as the name if that name is free; otherwise, it appends a string such as '`<2>`' to get an unused name. See also Section 16.9 [Creating Buffers], page 284.

**Please note:** `create-file-buffer` does *not* associate the new buffer with a file and does not select the buffer. It also does not use the default major mode.

```
(create-file-buffer "foo")
     ⇒ #<buffer foo>
(create-file-buffer "foo")
     ⇒ #<buffer foo<2>>
(create-file-buffer "foo")
     ⇒ #<buffer foo<3>>
```

This function is used by `find-file-noselect`. It uses `generate-new-buffer` (see Section 16.9 [Creating Buffers], page 284).

`after-find-file` **&optional** *error warn noauto* [Function]
*after-find-file-from-revert-buffer nomodes*

This function sets the buffer major mode, and parses local variables (see Section 20.2.2 [Auto Major Mode], page 403). It is called by `find-file-noselect` and by the default revert function (see ⟨undefined⟩ [Reverting], page ⟨undefined⟩).

If reading the file got an error because the file does not exist, but its directory does exist, the caller should pass a non-`nil` value for *error*. In that case, `after-find-file` issues a warning: '`(New file)`'. For more serious errors, the caller should usually not call `after-find-file`.

If *warn* is non-`nil`, then this function issues a warning if an auto-save file exists and is more recent than the visited file.

If *noauto* is non-`nil`, that says not to enable or disable Auto-Save mode. The mode remains enabled if it was enabled before.

If *after-find-file-from-revert-buffer* is non-`nil`, that means this call was from `revert-buffer`. This has no direct effect, but some mode functions and hook functions check the value of this variable.

If *nomodes* is non-`nil`, that means don't alter the buffer's major mode, don't process local variables specifications in the file, and don't run `find-file-hook`. This feature is used by `revert-buffer` in some cases.

The last thing `after-find-file` does is call all the functions in the list `find-file-hook`.

## 15.2 Saving Buffers

When you edit a file in Emacs, you are actually working on a buffer that is visiting that file—that is, the contents of the file are copied into the buffer and the copy is what you edit. Changes to the buffer do not change the file until you *save* the buffer, which means copying the contents of the buffer into the file.

`save-buffer` **&optional** *backup-option* [Command]

This function saves the contents of the current buffer in its visited file if the buffer has been modified since it was last visited or saved. Otherwise it does nothing.

`save-buffer` is responsible for making backup files. Normally, *backup-option* is `nil`, and `save-buffer` makes a backup file only if this is the first save since visiting the file. Other values for *backup-option* request the making of backup files in other circumstances:

- With an argument of 4 or 64, reflecting 1 or 3 `C-u`'s, the `save-buffer` function marks this version of the file to be backed up when the buffer is next saved.
- With an argument of 16 or 64, reflecting 2 or 3 `C-u`'s, the `save-buffer` function unconditionally backs up the previous version of the file before saving it.
- With an argument of 0, unconditionally do *not* make any backup file.

`save-some-buffers` **&optional** *save-silently-p pred* [Command]

This command saves some modified file-visiting buffers. Normally it asks the user about each buffer. But if *save-silently-p* is non-`nil`, it saves all the file-visiting buffers without querying the user.

The optional *pred* argument controls which buffers to ask about (or to save silently if *save-silently-p* is non-`nil`). If it is `nil`, that means to ask only about file-visiting buffers. If it is `t`, that means also offer to save certain other non-file buffers—those that have a non-`nil` buffer-local value of `buffer-offer-save` (see Section 16.10 [Killing Buffers], page 284). A user who says '`yes`' to saving a non-file buffer is asked to specify the file name to use. The `save-buffers-kill-emacs` function passes the value `t` for *pred*.

If *pred* is neither `t` nor `nil`, then it should be a function of no arguments. It will be called in each buffer to decide whether to offer to save that buffer. If it returns a non-`nil` value in a certain buffer, that means do offer to save that buffer.

`write-file` *filename* **&optional** *confirm* [Command]

This function writes the current buffer into file *filename*, makes the buffer visit that file, and marks it not modified. Then it renames the buffer based on *filename*, appending a string like '`<2>`' if necessary to make a unique buffer name. It does most

of this work by calling `set-visited-file-name` (see Section 16.4 [Buffer File Name], page 276) and `save-buffer`.

If *confirm* is non-`nil`, that means to ask for confirmation before overwriting an existing file. Interactively, confirmation is required, unless the user supplies a prefix argument.

If *filename* is an existing directory, or a symbolic link to one, `write-file` uses the name of the visited file, in directory *filename*. If the buffer is not visiting a file, it uses the buffer name instead.

Saving a buffer runs several hooks. It also performs format conversion (see Section 15.12 [Format Conversion], page 267).

`write-file-functions` [Variable]

The value of this variable is a list of functions to be called before writing out a buffer to its visited file. If one of them returns non-`nil`, the file is considered already written and the rest of the functions are not called, nor is the usual code for writing the file executed.

If a function in `write-file-functions` returns non-`nil`, it is responsible for making a backup file (if that is appropriate). To do so, execute the following code:

```
(or buffer-backed-up (backup-buffer))
```

You might wish to save the file modes value returned by `backup-buffer` and use that (if non-`nil`) to set the mode bits of the file that you write. This is what `save-buffer` normally does. See ⟨undefined⟩ [Making Backup Files], page ⟨undefined⟩.

The hook functions in `write-file-functions` are also responsible for encoding the data (if desired): they must choose a suitable coding system and end-of-line conversion (see ⟨undefined⟩ [Lisp and Coding Systems], page ⟨undefined⟩), perform the encoding (see ⟨undefined⟩ [Explicit Encoding], page ⟨undefined⟩), and set `last-coding-system-used` to the coding system that was used (see ⟨undefined⟩ [Encoding and I/O], page ⟨undefined⟩).

If you set this hook locally in a buffer, it is assumed to be associated with the file or the way the contents of the buffer were obtained. Thus the variable is marked as a permanent local, so that changing the major mode does not alter a buffer-local value. On the other hand, calling `set-visited-file-name` will reset it. If this is not what you want, you might like to use `write-contents-functions` instead.

Even though this is not a normal hook, you can use `add-hook` and `remove-hook` to manipulate the list. See Section 33.2.2 [Hooks], page 696.

`write-contents-functions` [Variable]

This works just like `write-file-functions`, but it is intended for hooks that pertain to the buffer's contents, not to the particular visited file or its location. Such hooks are usually set up by major modes, as buffer-local bindings for this variable. This variable automatically becomes buffer-local whenever it is set; switching to a new major mode always resets this variable, but calling `set-visited-file-name` does not.

If any of the functions in this hook returns non-`nil`, the file is considered already written and the rest are not called and neither are the functions in `write-file-functions`.

`before-save-hook` [User Option]
: This normal hook runs before a buffer is saved in its visited file, regardless of whether that is done normally or by one of the hooks described above. For instance, the '`copyright.el`' program uses this hook to make sure the file you are saving has the current year in its copyright notice.

`after-save-hook` [User Option]
: This normal hook runs after a buffer has been saved in its visited file. One use of this hook is in Fast Lock mode; it uses this hook to save the highlighting information in a cache file.

`file-precious-flag` [User Option]
: If this variable is non-`nil`, then `save-buffer` protects against I/O errors while saving by writing the new file to a temporary name instead of the name it is supposed to have, and then renaming it to the intended name after it is clear there are no errors. This procedure prevents problems such as a lack of disk space from resulting in an invalid file.

  As a side effect, backups are necessarily made by copying. See ⟨undefined⟩ [Rename or Copy], page ⟨undefined⟩. Yet, at the same time, saving a precious file always breaks all hard links between the file you save and other file names.

  Some modes give this variable a non-`nil` buffer-local value in particular buffers.

`require-final-newline` [User Option]
: This variable determines whether files may be written out that do *not* end with a newline. If the value of the variable is `t`, then `save-buffer` silently adds a newline at the end of the buffer whenever it does not already end in one. If the value is `visit`, Emacs adds a missing newline just after it visits the file. If the value is `visit-save`, Emacs adds a missing newline both on visiting and on saving. For any other non-`nil` value, `save-buffer` asks the user whether to add a newline each time the case arises.

  If the value of the variable is `nil`, then `save-buffer` doesn't add newlines at all. `nil` is the default value, but a few major modes set it to `t` in particular buffers.

See also the function `set-visited-file-name` (see Section 16.4 [Buffer File Name], page 276).

## 15.3 Reading from Files

You can copy a file from the disk and insert it into a buffer using the `insert-file-contents` function. Don't use the user-level command `insert-file` in a Lisp program, as that sets the mark.

`insert-file-contents` *filename* **&optional** *visit beg end replace* [Function]
: This function inserts the contents of file *filename* into the current buffer after point. It returns a list of the absolute file name and the length of the data inserted. An error is signaled if *filename* is not the name of a file that can be read.

  This function checks the file contents against the defined file formats, and converts the file contents if appropriate and also calls the functions in the list `after-insert-file-functions`. See Section 15.12 [Format Conversion], page 267. Normally, one

of the functions in the `after-insert-file-functions` list determines the coding system (see Section 19.6 [Coding Systems], page 381) used for decoding the file's contents, including end-of-line conversion. However, if the file contains null bytes, it is by default visited without any code conversions. See ⟨undefined⟩ [Lisp and Coding Systems], page ⟨undefined⟩.

If *visit* is non-`nil`, this function additionally marks the buffer as unmodified and sets up various fields in the buffer so that it is visiting the file *filename*: these include the buffer's visited file name and its last save file modtime. This feature is used by `find-file-noselect` and you probably should not use it yourself.

If *beg* and *end* are non-`nil`, they should be integers specifying the portion of the file to insert. In this case, *visit* must be `nil`. For example,

```
(insert-file-contents filename nil 0 500)
```

inserts the first 500 characters of a file.

If the argument *replace* is non-`nil`, it means to replace the contents of the buffer (actually, just the accessible portion) with the contents of the file. This is better than simply deleting the buffer contents and inserting the whole file, because (1) it preserves some marker positions and (2) it puts less data in the undo list.

It is possible to read a special file (such as a FIFO or an I/O device) with `insert-file-contents`, as long as *replace* and *visit* are `nil`.

`insert-file-contents-literally` *filename* **&optional** *visit beg end replace* [Function]

This function works like `insert-file-contents` except that it does not run `find-file-hook`, and does not do format decoding, character code conversion, automatic uncompression, and so on.

If you want to pass a file name to another process so that another program can read the file, use the function `file-local-copy`; see Section 15.11 [Magic File Names], page 262.

## 15.4 Writing to Files

You can write the contents of a buffer, or part of a buffer, directly to a file on disk using the `append-to-file` and `write-region` functions. Don't use these functions to write to files that are being visited; that could cause confusion in the mechanisms for visiting.

`append-to-file` *start end filename* [Command]

This function appends the contents of the region delimited by *start* and *end* in the current buffer to the end of file *filename*. If that file does not exist, it is created. This function returns `nil`.

An error is signaled if *filename* specifies a nonwritable file, or a nonexistent file in a directory where files cannot be created.

When called from Lisp, this function is completely equivalent to:

```
(write-region start end filename t)
```

`write-region` *start end filename* **&optional** *append visit lockname mustbenew* [Command]

This function writes the region delimited by *start* and *end* in the current buffer into the file specified by *filename*.

If *start* is `nil`, then the command writes the entire buffer contents (*not* just the accessible portion) to the file and ignores *end*.

If *start* is a string, then `write-region` writes or appends that string, rather than text from the buffer. *end* is ignored in this case.

If *append* is non-`nil`, then the specified text is appended to the existing file contents (if any). If *append* is an integer, `write-region` seeks to that byte offset from the start of the file and writes the data from there.

If *mustbenew* is non-`nil`, then `write-region` asks for confirmation if *filename* names an existing file. If *mustbenew* is the symbol `excl`, then `write-region` does not ask for confirmation, but instead it signals an error `file-already-exists` if the file already exists.

The test for an existing file, when *mustbenew* is `excl`, uses a special system feature. At least for files on a local disk, there is no chance that some other program could create a file of the same name before Emacs does, without Emacs's noticing.

If *visit* is `t`, then Emacs establishes an association between the buffer and the file: the buffer is then visiting that file. It also sets the last file modification time for the current buffer to *filename*'s modtime, and marks the buffer as not modified. This feature is used by `save-buffer`, but you probably should not use it yourself.

If *visit* is a string, it specifies the file name to visit. This way, you can write the data to one file (*filename*) while recording the buffer as visiting another file (*visit*). The argument *visit* is used in the echo area message and also for file locking; *visit* is stored in `buffer-file-name`. This feature is used to implement `file-precious-flag`; don't use it yourself unless you really know what you're doing.

The optional argument *lockname*, if non-`nil`, specifies the file name to use for purposes of locking and unlocking, overriding *filename* and *visit* for that purpose.

The function `write-region` converts the data which it writes to the appropriate file formats specified by `buffer-file-format` and also calls the functions in the list `write-region-annotate-functions`. See Section 15.12 [Format Conversion], page 267.

Normally, `write-region` displays the message '`Wrote` *`filename`*' in the echo area. If *visit* is neither `t` nor `nil` nor a string, then this message is inhibited. This feature is useful for programs that use files for internal purposes, files that the user does not need to know about.

`with-temp-file` *file body...* [Macro]

The `with-temp-file` macro evaluates the *body* forms with a temporary buffer as the current buffer; then, at the end, it writes the buffer contents into file *file*. It kills the temporary buffer when finished, restoring the buffer that was current before the `with-temp-file` form. Then it returns the value of the last form in *body*.

The current buffer is restored even in case of an abnormal exit via `throw` or error (see ⟨undefined⟩ [Nonlocal Exits], page ⟨undefined⟩).

See also `with-temp-buffer` in [The Current Buffer], page 274.

## 15.5 File Locks

When two users edit the same file at the same time, they are likely to interfere with each other. Emacs tries to prevent this situation from arising by recording a *file lock* when a file is being modified. (File locks are not implemented on Microsoft systems.) Emacs can then detect the first attempt to modify a buffer visiting a file that is locked by another Emacs job, and ask the user what to do. The file lock is really a file, a symbolic link with a special name, stored in the same directory as the file you are editing.

When you access files using NFS, there may be a small probability that you and another user will both lock the same file "simultaneously". If this happens, it is possible for the two users to make changes simultaneously, but Emacs will still warn the user who saves second. Also, the detection of modification of a buffer visiting a file changed on disk catches some cases of simultaneous editing; see Section 16.6 [Modification Time], page 279.

`file-locked-p` *filename* [Function]
: This function returns `nil` if the file *filename* is not locked. It returns `t` if it is locked by this Emacs process, and it returns the name of the user who has locked it if it is locked by some other job.

```
(file-locked-p "foo")
     ⇒ nil
```

`lock-buffer` **&optional** *filename* [Function]
: This function locks the file *filename*, if the current buffer is modified. The argument *filename* defaults to the current buffer's visited file. Nothing is done if the current buffer is not visiting a file, or is not modified, or if the system does not support locking.

`unlock-buffer` [Function]
: This function unlocks the file being visited in the current buffer, if the buffer is modified. If the buffer is not modified, then the file should not be locked, so this function does nothing. It also does nothing if the current buffer is not visiting a file, or if the system does not support locking.

File locking is not supported on some systems. On systems that do not support it, the functions `lock-buffer`, `unlock-buffer` and `file-locked-p` do nothing and return `nil`. It is also possible to disable locking, by setting the variable `create-lockfiles`.

`create-lockfiles` [User Option]
: If this variable is `nil`, Emacs does not lock files.

`ask-user-about-lock` *file other-user* [Function]
: This function is called when the user tries to modify *file*, but it is locked by another user named *other-user*. The default definition of this function asks the user to say what to do. The value this function returns determines what Emacs does next:

  - A value of `t` says to grab the lock on the file. Then this user may edit the file and *other-user* loses the lock.
  - A value of `nil` says to ignore the lock and let this user edit the file anyway.

- This function may instead signal a `file-locked` error, in which case the change that the user was about to make does not take place.

  The error message for this error looks like this:

  error `File is locked: file other-user`

  where `file` is the name of the file and *other-user* is the name of the user who has locked the file.

If you wish, you can replace the `ask-user-about-lock` function with your own version that makes the decision in another way. The code for its usual definition is in '`userlock.el`'.

## 15.6 Information about Files

The functions described in this section all operate on strings that designate file names. With a few exceptions, all the functions have names that begin with the word '`file`'. These functions all return information about actual files or directories, so their arguments must all exist as actual files or directories unless otherwise noted.

### 15.6.1 Testing Accessibility

These functions test for permission to access a file in specific ways. Unless explicitly stated otherwise, they recursively follow symbolic links for their file name arguments, at all levels (at the level of the file itself and at all levels of parent directories).

`file-exists-p` *filename* [Function]

This function returns `t` if a file named *filename* appears to exist. This does not mean you can necessarily read the file, only that you can find out its attributes. (On Unix and GNU/Linux, this is true if the file exists and you have execute permission on the containing directories, regardless of the permissions of the file itself.)

If the file does not exist, or if fascist access control policies prevent you from finding the attributes of the file, this function returns `nil`.

Directories are files, so `file-exists-p` returns `t` when given a directory name. However, symbolic links are treated specially; `file-exists-p` returns `t` for a symbolic link name only if the target file exists.

`file-readable-p` *filename* [Function]

This function returns `t` if a file named *filename* exists and you can read it. It returns `nil` otherwise.

```
(file-readable-p "files.texi")
     ⇒ t
(file-exists-p "/usr/spool/mqueue")
     ⇒ t
(file-readable-p "/usr/spool/mqueue")
     ⇒ nil
```

`file-executable-p` *filename* [Function]

This function returns `t` if a file named *filename* exists and you can execute it. It returns `nil` otherwise. On Unix and GNU/Linux, if the file is a directory, execute

permission means you can check the existence and attributes of files inside the directory, and open those files if their modes permit.

`file-writable-p` *filename* [Function]
This function returns `t` if the file *filename* can be written or created by you, and `nil` otherwise. A file is writable if the file exists and you can write it. It is creatable if it does not exist, but the specified directory does exist and you can write in that directory.

In the third example below, '`foo`' is not writable because the parent directory does not exist, even though the user could create such a directory.

```
(file-writable-p "~/foo")
     ⇒ t
(file-writable-p "/foo")
     ⇒ nil
(file-writable-p "~/no-such-dir/foo")
     ⇒ nil
```

`file-accessible-directory-p` *dirname* [Function]
This function returns `t` if you have permission to open existing files in the directory whose name as a file is *dirname*; otherwise (or if there is no such directory), it returns `nil`. The value of *dirname* may be either a directory name (such as '`/foo/`') or the file name of a file which is a directory (such as '`/foo`', without the final slash).

Example: after the following,

```
(file-accessible-directory-p "/foo")
     ⇒ nil
```

we can deduce that any attempt to read a file in '`/foo/`' will give an error.

`access-file` *filename string* [Function]
This function opens file *filename* for reading, then closes it and returns `nil`. However, if the open fails, it signals an error using *string* as the error message text.

`file-ownership-preserved-p` *filename* [Function]
This function returns `t` if deleting the file *filename* and then creating it anew would keep the file's owner unchanged. It also returns `t` for nonexistent files.

If *filename* is a symbolic link, then, unlike the other functions discussed here, `file-ownership-preserved-p` does *not* replace *filename* with its target. However, it does recursively follow symbolic links at all levels of parent directories.

`file-newer-than-file-p` *filename1 filename2* [Function]
This function returns `t` if the file *filename1* is newer than file *filename2*. If *filename1* does not exist, it returns `nil`. If *filename1* does exist, but *filename2* does not, it returns `t`.

In the following example, assume that the file '`aug-19`' was written on the 19th, '`aug-20`' was written on the 20th, and the file '`no-file`' doesn't exist at all.

```
(file-newer-than-file-p "aug-19" "aug-20")
     ⇒ nil
```

```
(file-newer-than-file-p "aug-20" "aug-19")
     ⇒ t
(file-newer-than-file-p "aug-19" "no-file")
     ⇒ t
(file-newer-than-file-p "no-file" "aug-19")
     ⇒ nil
```

You can use `file-attributes` to get a file's last modification time as a list of four integers. See Section 15.6.4 [File Attributes], page 244.

### 15.6.2 Distinguishing Kinds of Files

This section describes how to distinguish various kinds of files, such as directories, symbolic links, and ordinary files.

`file-symlink-p` *filename* [Function]

If the file *filename* is a symbolic link, the `file-symlink-p` function returns the (non-recursive) link target as a string. (Determining the file name that the link points to from the target is nontrivial.) First, this function recursively follows symbolic links at all levels of parent directories.

If the file *filename* is not a symbolic link (or there is no such file), `file-symlink-p` returns `nil`.

```
(file-symlink-p "foo")
     ⇒ nil
(file-symlink-p "sym-link")
     ⇒ "foo"
(file-symlink-p "sym-link2")
     ⇒ "sym-link"
(file-symlink-p "/bin")
     ⇒ "/pub/bin"
```

The next two functions recursively follow symbolic links at all levels for *filename*.

`file-directory-p` *filename* [Function]

This function returns `t` if *filename* is the name of an existing directory, `nil` otherwise.

```
(file-directory-p "~rms")
     ⇒ t
(file-directory-p "~rms/lewis/files.texi")
     ⇒ nil
(file-directory-p "~rms/lewis/no-such-file")
     ⇒ nil
(file-directory-p "$HOME")
     ⇒ nil
(file-directory-p
 (substitute-in-file-name "$HOME"))
     ⇒ t
```

`file-regular-p` *filename* [Function]

This function returns `t` if the file *filename* exists and is a regular file (not a directory, named pipe, terminal, or other I/O device).

`file-equal-p` *file1 file2* [Function]
: This function returns `t` if the files *file1* and *file2* name the same file. If *file1* or *file2* does not exist, the return value is unspecified.

`file-in-directory-p` *file dir* [Function]
: This function returns `t` if *file* is a file in directory *dir*, or in a subdirectory of *dir*. It also returns `t` if *file* and *dir* are the same directory. It compares the `file-truename` values of the two directories (see Section 15.6.3 [Truenames], page 243). If *dir* does not name an existing directory, the return value is `nil`.

### 15.6.3 Truenames

The *truename* of a file is the name that you get by following symbolic links at all levels until none remain, then simplifying away '`.`' and '`..`' appearing as name components. This results in a sort of canonical name for the file. A file does not always have a unique truename; the number of distinct truenames a file has is equal to the number of hard links to the file. However, truenames are useful because they eliminate symbolic links as a cause of name variation.

`file-truename` *filename* [Function]
: This function returns the truename of the file *filename*. If the argument is not an absolute file name, this function first expands it against `default-directory`.

  This function does not expand environment variables. Only `substitute-in-file-name` does that. See [Definition of substitute-in-file-name], page 256.

  If you may need to follow symbolic links preceding '`..`' appearing as a name component, you should make sure to call `file-truename` without prior direct or indirect calls to `expand-file-name`, as otherwise the file name component immediately preceding '`..`' will be "simplified away" before `file-truename` is called. To eliminate the need for a call to `expand-file-name`, `file-truename` handles '`~`' in the same way that `expand-file-name` does. See Section 15.8.4 [Functions that Expand Filenames], page 255.

`file-chase-links` *filename* **&optional** *limit* [Function]
: This function follows symbolic links, starting with *filename*, until it finds a file name which is not the name of a symbolic link. Then it returns that file name. This function does *not* follow symbolic links at the level of parent directories.

  If you specify a number for *limit*, then after chasing through that many links, the function just returns what it has even if that is still a symbolic link.

To illustrate the difference between `file-chase-links` and `file-truename`, suppose that '`/usr/foo`' is a symbolic link to the directory '`/home/foo`', and '`/home/foo/hello`' is an ordinary file (or at least, not a symbolic link) or nonexistent. Then we would have:

```
(file-chase-links "/usr/foo/hello")
     ;; This does not follow the links in the parent directories.
     ⇒ "/usr/foo/hello"
(file-truename "/usr/foo/hello")
     ;; Assuming that '/home' is not a symbolic link.
     ⇒ "/home/foo/hello"
```

See Section 16.4 [Buffer File Name], page 276, for related information.

### 15.6.4 Other Information about Files

This section describes the functions for getting detailed information about a file, other than its contents. This information includes the mode bits that control access permissions, the owner and group numbers, the number of names, the inode number, the size, and the times of access and modification.

`file-modes` *filename* [Function]

This function returns the *mode bits* describing the *file permissions* of *filename*, as an integer. It recursively follows symbolic links in *filename* at all levels. If *filename* does not exist, the return value is `nil`.

See Section "File Permissions" in *The* GNU `Coreutils` *Manual*, for a description of mode bits. If the low-order bit is 1, then the file is executable by all users, if the second-lowest-order bit is 1, then the file is writable by all users, etc. The highest value returnable is 4095 (7777 octal), meaning that everyone has read, write, and execute permission, that the SUID bit is set for both others and group, and that the sticky bit is set.

```
(file-modes "~/junk/diffs")
     ⇒ 492               ; Decimal integer.
(format "%o" 492)
     ⇒ "754"             ; Convert to octal.

(set-file-modes "~/junk/diffs" #o666)
     ⇒ nil

% ls -l diffs
  -rw-rw-rw-  1 lewis 0 3063 Oct 30 16:00 diffs
```

See Section 15.7 [Changing Files], page 248, for functions that change file permissions, such as `set-file-modes`.

**MS-DOS note:** On MS-DOS, there is no such thing as an "executable" file mode bit. So `file-modes` considers a file executable if its name ends in one of the standard executable extensions, such as '`.com`', '`.bat`', '`.exe`', and some others. Files that begin with the Unix-standard '`#!`' signature, such as shell and Perl scripts, are also considered executable. Directories are also reported as executable, for compatibility with Unix. These conventions are also followed by `file-attributes`, below.

If the *filename* argument to the next two functions is a symbolic link, then these function do *not* replace it with its target. However, they both recursively follow symbolic links at all levels of parent directories.

`file-nlinks` *filename* [Function]

This functions returns the number of names (i.e., hard links) that file *filename* has. If the file does not exist, then this function returns `nil`. Note that symbolic links have no effect on this function, because they are not considered to be names of the files they link to.

```
% ls -l foo*
-rw-rw-rw-  2 rms       4 Aug 19 01:27 foo
-rw-rw-rw-  2 rms       4 Aug 19 01:27 foo1
```

```
(file-nlinks "foo")
     ⇒ 2
(file-nlinks "doesnt-exist")
     ⇒ nil
```

`file-attributes` *filename* **&optional** *id-format* [Function]

This function returns a list of attributes of file *filename*. If the specified file cannot be opened, it returns `nil`. The optional parameter *id-format* specifies the preferred format of attributes UID and GID (see below)—the valid values are `'string` and `'integer`. The latter is the default, but we plan to change that, so you should specify a non-`nil` value for *id-format* if you use the returned UID or GID.

The elements of the list, in order, are:

0. `t` for a directory, a string for a symbolic link (the name linked to), or `nil` for a text file.
1. The number of names the file has. Alternate names, also known as hard links, can be created by using the `add-name-to-file` function (see Section 15.7 [Changing Files], page 248).
2. The file's UID, normally as a string. However, if it does not correspond to a named user, the value is an integer or a floating point number.
3. The file's GID, likewise.
4. The time of last access, as a list of four integers `(`*`sec-high sec-low microsec picosec`*`)`. (This is similar to the value of `current-time`; see ⟨undefined⟩ [Time of Day], page ⟨undefined⟩.) Note that on some FAT-based filesystems, only the date of last access is recorded, so this time will always hold the midnight of the day of last access.
5. The time of last modification as a list of four integers (as above). This is the last time when the file's contents were modified.
6. The time of last status change as a list of four integers (as above). This is the time of the last change to the file's access mode bits, its owner and group, and other information recorded in the filesystem for the file, beyond the file's contents.
7. The size of the file in bytes. If the size is too large to fit in a Lisp integer, this is a floating point number.
8. The file's modes, as a string of ten letters or dashes, as in '`ls -l`'.
9. `t` if the file's GID would change if file were deleted and recreated; `nil` otherwise.
10. The file's inode number. If possible, this is an integer. If the inode number is too large to be represented as an integer in Emacs Lisp but dividing it by $2^16$ yields a representable integer, then the value has the form `(`*`high . low`*`)`, where *low* holds the low 16 bits. If the inode number is too wide for even that, the value is of the form `(`*`high middle . low`*`)`, where `high` holds the high bits, *middle* the middle 24 bits, and *low* the low 16 bits.
11. The filesystem number of the device that the file is on. Depending on the magnitude of the value, this can be either an integer or a cons cell, in the same manner as the inode number. This element and the file's inode number together give

enough information to distinguish any two files on the system—no two files can have the same values for both of these numbers.

For example, here are the file attributes for '`files.texi`':

```
(file-attributes "files.texi" 'string)
     ⇒  (nil 1 "lh" "users"
         (20614 64019 50040 152000)
         (20000 23 0 0)
         (20614 64555 902289 872000)
         122295 "-rw-rw-rw-"
         nil  (5888 2 . 43978)
         (15479 . 46724))
```

and here is how the result is interpreted:

`nil` is neither a directory nor a symbolic link.

`1` has only one name (the name '`files.texi`' in the current default directory).

`"lh"` is owned by the user with name "lh".

`"users"` is in the group with name "users".

`(20614 64019 50040 152000)`
was last accessed on October 23, 2012, at 20:12:03.050040152 UTC.

`(20000 23 0 0)`
was last modified on July 15, 2001, at 08:53:43 UTC.

`(20614 64555 902289 872000)`
last had its status changed on October 23, 2012, at 20:20:59.902289872 UTC.

`122295` is 122295 bytes long. (It may not contain 122295 characters, though, if some of the bytes belong to multibyte sequences, and also if the end-of-line format is CR-LF.)

`"-rw-rw-rw-"`
has a mode of read and write access for the owner, group, and world.

`nil` would retain the same GID if it were recreated.

`(5888 2 . 43978)`
has an inode number of 6473924464520138.

`(15479 . 46724)`
is on the file-system device whose number is 1014478468.

SELinux is a Linux kernel feature which provides more sophisticated file access controls than ordinary "Unix-style" file permissions. If Emacs has been compiled with SELinux support on a system with SELinux enabled, you can use the function `file-selinux-context` to retrieve a file's SELinux security context. For the function `set-file-selinux-context`, see Section 15.7 [Changing Files], page 248.

`file-selinux-context` *filename* [Function]
: This function returns the SELinux security context of the file *filename*. This return value is a list of the form `(user role type range)`, whose elements are the context's user, role, type, and range respectively, as Lisp strings. See the SELinux documentation for details about what these actually mean.

  If the file does not exist or is inaccessible, or if the system does not support SELinux, or if Emacs was not compiled with SELinux support, then the return value is `(nil nil nil nil)`.

### 15.6.5 How to Locate Files in Standard Places

This section explains how to search for a file in a list of directories (a *path*), or for an executable file in the standard list of executable file directories.

To search for a user-specific configuration file, See Section 15.8.7 [Standard File Names], page 259, for the `locate-user-emacs-file` function.

`locate-file` *filename path* **&optional** *suffixes predicate* [Function]
: This function searches for a file whose name is *filename* in a list of directories given by *path*, trying the suffixes in *suffixes*. If it finds such a file, it returns the file's absolute file name (see Section 15.8.2 [Relative File Names], page 253); otherwise it returns `nil`.

  The optional argument *suffixes* gives the list of file-name suffixes to append to *filename* when searching. `locate-file` tries each possible directory with each of these suffixes. If *suffixes* is `nil`, or `("")`, then there are no suffixes, and *filename* is used only as-is. Typical values of *suffixes* are `exec-suffixes` (see ⟨undefined⟩ [Subprocess Creation], page ⟨undefined⟩), `load-suffixes`, `load-file-rep-suffixes` and the return value of the function `get-load-suffixes` (see ⟨undefined⟩ [Load Suffixes], page ⟨undefined⟩).

  Typical values for *path* are `exec-path` (see ⟨undefined⟩ [Subprocess Creation], page ⟨undefined⟩) when looking for executable programs, or `load-path` (see ⟨undefined⟩ [Library Search], page ⟨undefined⟩) when looking for Lisp files. If *filename* is absolute, *path* has no effect, but the suffixes in *suffixes* are still tried.

  The optional argument *predicate*, if non-`nil`, specifies a predicate function for testing whether a candidate file is suitable. The predicate is passed the candidate file name as its single argument. If *predicate* is `nil` or omitted, `locate-file` uses `file-readable-p` as the predicate. See Section 15.6.2 [Kinds of Files], page 242, for other useful predicates, e.g., `file-executable-p` and `file-directory-p`.

  For compatibility, *predicate* can also be one of the symbols `executable`, `readable`, `writable`, `exists`, or a list of one or more of these symbols.

`executable-find` *program* [Function]
: This function searches for the executable file of the named *program* and returns the absolute file name of the executable, including its file-name extensions, if any. It returns `nil` if the file is not found. The functions searches in all the directories in `exec-path`, and tries all the file-name extensions in `exec-suffixes` (see ⟨undefined⟩ [Subprocess Creation], page ⟨undefined⟩).

## 15.7 Changing File Names and Attributes

The functions in this section rename, copy, delete, link, and set the modes (permissions) of files.

In the functions that have an argument *newname*, if a file by the name of *newname* already exists, the actions taken depend on the value of the argument *ok-if-already-exists*:

- Signal a `file-already-exists` error if *ok-if-already-exists* is `nil`.
- Request confirmation if *ok-if-already-exists* is a number.
- Replace the old file without confirmation if *ok-if-already-exists* is any other value.

The next four commands all recursively follow symbolic links at all levels of parent directories for their first argument, but, if that argument is itself a symbolic link, then only `copy-file` replaces it with its (recursive) target.

`add-name-to-file` *oldname newname* **&optional** *ok-if-already-exists* [Command]

This function gives the file named *oldname* the additional name *newname*. This means that *newname* becomes a new "hard link" to *oldname*.

In the first part of the following example, we list two files, '`foo`' and '`foo3`'.

```
% ls -li fo*
81908 -rw-rw-rw-  1 rms       29 Aug 18 20:32 foo
84302 -rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

Now we create a hard link, by calling `add-name-to-file`, then list the files again. This shows two names for one file, '`foo`' and '`foo2`'.

```
(add-name-to-file "foo" "foo2")
     ⇒ nil

% ls -li fo*
81908 -rw-rw-rw-  2 rms       29 Aug 18 20:32 foo
81908 -rw-rw-rw-  2 rms       29 Aug 18 20:32 foo2
84302 -rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

Finally, we evaluate the following:

```
(add-name-to-file "foo" "foo3" t)
```

and list the files again. Now there are three names for one file: '`foo`', '`foo2`', and '`foo3`'. The old contents of '`foo3`' are lost.

```
(add-name-to-file "foo1" "foo3")
     ⇒ nil

% ls -li fo*
81908 -rw-rw-rw-  3 rms       29 Aug 18 20:32 foo
81908 -rw-rw-rw-  3 rms       29 Aug 18 20:32 foo2
81908 -rw-rw-rw-  3 rms       29 Aug 18 20:32 foo3
```

This function is meaningless on operating systems where multiple names for one file are not allowed. Some systems implement multiple names by copying the file instead.

See also `file-nlinks` in Section 15.6.4 [File Attributes], page 244.

`rename-file` *filename newname* **&optional** *ok-if-already-exists* [Command]
This command renames the file *filename* as *newname*.

If *filename* has additional names aside from *filename*, it continues to have those names. In fact, adding the name *newname* with `add-name-to-file` and then deleting *filename* has the same effect as renaming, aside from momentary intermediate states.

`copy-file` *oldname newname* **&optional** *ok-if-exists time preserve-uid-gid preserve-selinux* [Command]
This command copies the file *oldname* to *newname*. An error is signaled if *oldname* does not exist. If *newname* names a directory, it copies *oldname* into that directory, preserving its final name component.

If *time* is non-`nil`, then this function gives the new file the same last-modified time that the old one has. (This works on only some operating systems.) If setting the time gets an error, `copy-file` signals a `file-date-error` error. In an interactive call, a prefix argument specifies a non-`nil` value for *time*.

This function copies the file modes, too.

If argument *preserve-uid-gid* is `nil`, we let the operating system decide the user and group ownership of the new file (this is usually set to the user running Emacs). If *preserve-uid-gid* is non-`nil`, we attempt to copy the user and group ownership of the file. This works only on some operating systems, and only if you have the correct permissions to do so.

If the optional argument *preserve-selinux* is non-`nil`, and Emacs has been compiled with SELinux support, this function attempts to copy the file's SELinux context (see Section 15.6.4 [File Attributes], page 244).

`make-symbolic-link` *filename newname* **&optional** *ok-if-exists* [Command]
This command makes a symbolic link to *filename*, named *newname*. This is like the shell command '`ln -s` *`filename newname`*'.

This function is not available on systems that don't support symbolic links.

`delete-file` *filename* **&optional** *trash* [Command]
This command deletes the file *filename*. If the file has multiple names, it continues to exist under the other names. If *filename* is a symbolic link, `delete-file` deletes only the symbolic link and not its target (though it does follow symbolic links at all levels of parent directories).

A suitable kind of `file-error` error is signaled if the file does not exist, or is not deletable. (On Unix and GNU/Linux, a file is deletable if its directory is writable.)

If the optional argument *trash* is non-`nil` and the variable `delete-by-moving-to-trash` is non-`nil`, this command moves the file into the system Trash instead of deleting it. See Section "Miscellaneous File Operations" in *The GNU Emacs Manual*. When called interactively, *trash* is `t` if no prefix argument is given, and `nil` otherwise.

See also `delete-directory` in Section 15.10 [Create/Delete Dirs], page 262.

`set-file-modes` *filename mode* [Command]
This function sets the *file mode* (or *file permissions*) of *filename* to *mode*. It recursively follows symbolic links at all levels for *filename*.

If called non-interactively, *mode* must be an integer. Only the lowest 12 bits of the integer are used; on most systems, only the lowest 9 bits are meaningful. You can use the Lisp construct for octal numbers to enter *mode*. For example,

```
(set-file-modes #o644)
```

specifies that the file should be readable and writable for its owner, readable for group members, and readable for all other users. See Section "File Permissions" in *The* GNU `Coreutils` *Manual*, for a description of mode bit specifications.

Interactively, *mode* is read from the minibuffer using `read-file-modes` (see below), which lets the user type in either an integer or a string representing the permissions symbolically.

See Section 15.6.4 [File Attributes], page 244, for the function `file-modes`, which returns the permissions of a file.

`set-default-file-modes` *mode* [Function]

This function sets the default file permissions for new files created by Emacs and its subprocesses. Every file created with Emacs initially has these permissions, or a subset of them (`write-region` will not grant execute permissions even if the default file permissions allow execution). On Unix and GNU/Linux, the default permissions are given by the bitwise complement of the "umask" value.

The argument *mode* should be an integer which specifies the permissions, similar to `set-file-modes` above. Only the lowest 9 bits are meaningful.

The default file permissions have no effect when you save a modified version of an existing file; saving a file preserves its existing permissions.

`default-file-modes` [Function]

This function returns the default file permissions, as an integer.

`read-file-modes` **&optional** *prompt base-file* [Function]

This function reads a set of file mode bits from the minibuffer. The first optional argument *prompt* specifies a non-default prompt. Second second optional argument *base-file* is the name of a file on whose permissions to base the mode bits that this function returns, if what the user types specifies mode bits relative to permissions of an existing file.

If user input represents an octal number, this function returns that number. If it is a complete symbolic specification of mode bits, as in `"u=rwx"`, the function converts it to the equivalent numeric value using `file-modes-symbolic-to-number` and returns the result. If the specification is relative, as in `"o+g"`, then the permissions on which the specification is based are taken from the mode bits of *base-file*. If *base-file* is omitted or `nil`, the function uses `0` as the base mode bits. The complete and relative specifications can be combined, as in `"u+r,g+rx,o+r,g-w"`. See Section "File Permissions" in *The* GNU `Coreutils` *Manual*, for a description of file mode specifications.

`file-modes-symbolic-to-number` *modes* **&optional** *base-modes* [Function]

This function converts a symbolic file mode specification in *modes* into the equivalent integer value. If the symbolic specification is based on an existing file, that file's mode bits are taken from the optional argument *base-modes*; if that argument is omitted or `nil`, it defaults to 0, i.e., no access rights at all.

`set-file-times` *filename* **&optional** *time* [Function]
: This function sets the access and modification times of *filename* to *time*. The return value is `t` if the times are successfully set, otherwise it is `nil`. *time* defaults to the current time and must be in the format returned by `current-time` (see ⟨undefined⟩ [Time of Day], page ⟨undefined⟩).

`set-file-selinux-context` *filename context* [Function]
: This function sets the SELinux security context of the file *filename* to *context*. See Section 15.6.4 [File Attributes], page 244, for a brief description of SELinux contexts. The *context* argument should be a list `(user role type range)`, like the return value of `file-selinux-context`. The function does nothing if SELinux is disabled, or if Emacs was compiled without SELinux support.

## 15.8 File Names

Files are generally referred to by their names, in Emacs as elsewhere. File names in Emacs are represented as strings. The functions that operate on a file all expect a file name argument.

In addition to operating on files themselves, Emacs Lisp programs often need to operate on file names; i.e., to take them apart and to use part of a name to construct related file names. This section describes how to manipulate file names.

The functions in this section do not actually access files, so they can operate on file names that do not refer to an existing file or directory.

On MS-DOS and MS-Windows, these functions (like the function that actually operate on files) accept MS-DOS or MS-Windows file-name syntax, where backslashes separate the components, as well as Unix syntax; but they always return Unix syntax. This enables Lisp programs to specify file names in Unix syntax and work properly on all systems without change.[1]

### 15.8.1 File Name Components

The operating system groups files into directories. To specify a file, you must specify the directory and the file's name within that directory. Therefore, Emacs considers a file name as having two main parts: the *directory name* part, and the *nondirectory* part (or *file name within the directory*). Either part may be empty. Concatenating these two parts reproduces the original file name.

On most systems, the directory part is everything up to and including the last slash (backslash is also allowed in input on MS-DOS or MS-Windows); the nondirectory part is the rest.

For some purposes, the nondirectory part is further subdivided into the name proper and the *version number*. On most systems, only backup files have version numbers in their names.

[1] In MS-Windows versions of Emacs compiled for the Cygwin environment, you can use the functions `cygwin-convert-file-name-to-windows` and `cygwin-convert-file-name-from-windows` to convert between the two file-name syntaxes.

`file-name-directory` *filename* [Function]
This function returns the directory part of *filename*, as a directory name (see Section 15.8.3 [Directory Names], page 254), or `nil` if *filename* does not include a directory part.

On GNU and Unix systems, a string returned by this function always ends in a slash. On MS-DOS it can also end in a colon.

```
(file-name-directory "lewis/foo")  ; Unix example
     ⇒ "lewis/"
(file-name-directory "foo")        ; Unix example
     ⇒ nil
```

`file-name-nondirectory` *filename* [Function]
This function returns the nondirectory part of *filename*.

```
(file-name-nondirectory "lewis/foo")
     ⇒ "foo"
(file-name-nondirectory "foo")
     ⇒ "foo"
(file-name-nondirectory "lewis/")
     ⇒ ""
```

`file-name-sans-versions` *filename* **&optional** *keep-backup-version* [Function]
This function returns *filename* with any file version numbers, backup version numbers, or trailing tildes discarded.

If *keep-backup-version* is non-`nil`, then true file version numbers understood as such by the file system are discarded from the return value, but backup version numbers are kept.

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
```

`file-name-extension` *filename* **&optional** *period* [Function]
This function returns *filename*'s final "extension", if any, after applying `file-name-sans-versions` to remove any version/backup part. The extension, in a file name, is the part that follows the last '`.`' in the last name component (minus any version/backup part).

This function returns `nil` for extensionless file names such as '`foo`'. It returns `""` for null extensions, as in '`foo.`'. If the last component of a file name begins with a '`.`', that '`.`' doesn't count as the beginning of an extension. Thus, '`.emacs`''s "extension" is `nil`, not '`.emacs`'.

If *period* is non-`nil`, then the returned value includes the period that delimits the extension, and if *filename* has no extension, the value is `""`.

`file-name-sans-extension` *filename* [Function]
This function returns *filename* minus its extension, if any. The version/backup part, if present, is only removed if the file has an extension. For example,

```
(file-name-sans-extension "foo.lose.c")
     ⇒ "foo.lose"
(file-name-sans-extension "big.hack/foo")
     ⇒ "big.hack/foo"
(file-name-sans-extension "/my/home/.emacs")
     ⇒ "/my/home/.emacs"
(file-name-sans-extension "/my/home/.emacs.el")
     ⇒ "/my/home/.emacs"
(file-name-sans-extension "~/foo.el.~3~")
     ⇒ "~/foo"
(file-name-sans-extension "~/foo.~3~")
     ⇒ "~/foo.~3~"
```

Note that the '`.~3~`' in the two last examples is the backup part, not an extension.

`file-name-base` **&optional** *filename* [Function]

This function is the composition of `file-name-sans-extension` and `file-name-nondirectory`. For example,

```
(file-name-base "/my/home/foo.c")
    ⇒ "foo"
```

The *filename* argument defaults to `buffer-file-name`.

### 15.8.2 Absolute and Relative File Names

All the directories in the file system form a tree starting at the root directory. A file name can specify all the directory names starting from the root of the tree; then it is called an *absolute* file name. Or it can specify the position of the file in the tree relative to a default directory; then it is called a *relative* file name. On Unix and GNU/Linux, an absolute file name starts with a '`/`' or a '`~`' (see [abbreviate-file-name], page 255), and a relative one does not. On MS-DOS and MS-Windows, an absolute file name starts with a slash or a backslash, or with a drive specification '`x:/`', where *x* is the *drive letter*.

`file-name-absolute-p` *filename* [Function]

This function returns `t` if file *filename* is an absolute file name, `nil` otherwise.

```
(file-name-absolute-p "~rms/foo")
     ⇒ t
(file-name-absolute-p "rms/foo")
     ⇒ nil
(file-name-absolute-p "/user/rms/foo")
     ⇒ t
```

Given a possibly relative file name, you can convert it to an absolute name using `expand-file-name` (see Section 15.8.4 [File Name Expansion], page 255). This function converts absolute file names to relative names:

`file-relative-name` *filename* **&optional** *directory* [Function]

This function tries to return a relative name that is equivalent to *filename*, assuming the result will be interpreted relative to *directory* (an absolute directory name or directory file name). If *directory* is omitted or `nil`, it defaults to the current buffer's default directory.

On some operating systems, an absolute file name begins with a device name. On such systems, *filename* has no relative equivalent based on *directory* if they start with two different device names. In this case, `file-relative-name` returns *filename* in absolute form.

```
(file-relative-name "/foo/bar" "/foo/")
     ⇒ "bar"
(file-relative-name "/foo/bar" "/hack/")
     ⇒ "../foo/bar"
```

### 15.8.3 Directory Names

A *directory name* is the name of a directory. A directory is actually a kind of file, so it has a file name, which is related to the directory name but not identical to it. (This is not quite the same as the usual Unix terminology.) These two different names for the same entity are related by a syntactic transformation. On GNU and Unix systems, this is simple: a directory name ends in a slash, whereas the directory's name as a file lacks that slash. On MS-DOS the relationship is more complicated.

The difference between a directory name and its name as a file is subtle but crucial. When an Emacs variable or function argument is described as being a directory name, a file name of a directory is not acceptable. When `file-name-directory` returns a string, that is always a directory name.

The following two functions convert between directory names and file names. They do nothing special with environment variable substitutions such as '`$HOME`', and the constructs '`~`', '`.`' and '`..`'.

`file-name-as-directory` *filename* [Function]

This function returns a string representing *filename* in a form that the operating system will interpret as the name of a directory. On most systems, this means appending a slash to the string (if it does not already end in one).

```
(file-name-as-directory "~rms/lewis")
     ⇒ "~rms/lewis/"
```

`directory-file-name` *dirname* [Function]

This function returns a string representing *dirname* in a form that the operating system will interpret as the name of a file. On most systems, this means removing the final slash (or backslash) from the string.

```
(directory-file-name "~lewis/")
     ⇒ "~lewis"
```

Given a directory name, you can combine it with a relative file name using `concat`:

```
(concat dirname relfile)
```

Be sure to verify that the file name is relative before doing that. If you use an absolute file name, the results could be syntactically invalid or refer to the wrong file.

If you want to use a directory file name in making such a combination, you must first convert it to a directory name using `file-name-as-directory`:

```
(concat (file-name-as-directory dirfile) relfile)
```

Don't try concatenating a slash by hand, as in

```
;;; Wrong!
(concat dirfile "/" relfile)
```

because this is not portable. Always use `file-name-as-directory`.

To convert a directory name to its abbreviation, use this function:

`abbreviate-file-name` *filename* [Function]

This function returns an abbreviated form of *filename*. It applies the abbreviations specified in `directory-abbrev-alist` (see Section "File Aliases" in *The GNU Emacs Manual*), then substitutes '`~`' for the user's home directory if the argument names a file in the home directory or one of its subdirectories. If the home directory is a root directory, it is not replaced with '`~`', because this does not make the result shorter on many systems.

You can use this function for directory names and for file names, because it recognizes abbreviations even as part of the name.

### 15.8.4 Functions that Expand Filenames

*Expanding* a file name means converting a relative file name to an absolute one. Since this is done relative to a default directory, you must specify the default directory name as well as the file name to be expanded. It also involves expanding abbreviations like '`~/`' and eliminating redundancies like '`./`' and '*name*`/../`'.

`expand-file-name` *filename* **&optional** *directory* [Function]

This function converts *filename* to an absolute file name. If *directory* is supplied, it is the default directory to start with if *filename* is relative. (The value of *directory* should itself be an absolute directory name or directory file name; it may start with '`~`'.) Otherwise, the current buffer's value of `default-directory` is used. For example:

```
(expand-file-name "foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
     ⇒ "/xcssun/users/rms/foo"
(expand-file-name "foo" "/usr/spool/")
     ⇒ "/usr/spool/foo"
(expand-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/lewis/$HOME/foo"
```

If the part of the combined file name before the first slash is '`~`', it expands to the value of the `HOME` environment variable (usually your home directory). If the part before the first slash is '*~user*' and if *user* is a valid login name, it expands to *user*'s home directory.

Filenames containing '`.`' or '`..`' are simplified to their canonical form:

```
(expand-file-name "bar/../foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
```

In some cases, a leading '`..`' component can remain in the output:

```
(expand-file-name "../home" "/")
     ⇒ "/../home"
```

This is for the sake of filesystems that have the concept of a "superroot" above the root directory '/'. On other filesystems, '/../' is interpreted exactly the same as '/'.

Note that `expand-file-name` does *not* expand environment variables; only `substitute-in-file-name` does that.

Note also that `expand-file-name` does not follow symbolic links at any level. This results in a difference between the way `file-truename` and `expand-file-name` treat '..'. Assuming that '/tmp/bar' is a symbolic link to the directory '/tmp/foo/bar' we get:

```
(file-truename "/tmp/bar/../myfile")
     ⇒ "/tmp/foo/myfile"
(expand-file-name "/tmp/bar/../myfile")
     ⇒ "/tmp/myfile"
```

If you may need to follow symbolic links preceding '..', you should make sure to call `file-truename` without prior direct or indirect calls to `expand-file-name`. See Section 15.6.3 [Truenames], page 243.

`default-directory` [Variable]

The value of this buffer-local variable is the default directory for the current buffer. It should be an absolute directory name; it may start with '~'. This variable is buffer-local in every buffer.

`expand-file-name` uses the default directory when its second argument is `nil`.

The value is always a string ending with a slash.

```
default-directory
     ⇒ "/user/lewis/manual/"
```

`substitute-in-file-name` *filename* [Function]

This function replaces environment variable references in *filename* with the environment variable values. Following standard Unix shell syntax, '$' is the prefix to substitute an environment variable value. If the input contains '$$', that is converted to '$'; this gives the user a way to "quote" a '$'.

The environment variable name is the series of alphanumeric characters (including underscores) that follow the '$'. If the character following the '$' is a '{', then the variable name is everything up to the matching '}'.

Calling `substitute-in-file-name` on output produced by `substitute-in-file-name` tends to give incorrect results. For instance, use of '$$' to quote a single '$' won't work properly, and '$' in an environment variable's value could lead to repeated substitution. Therefore, programs that call this function and put the output where it will be passed to this function need to double all '$' characters to prevent subsequent incorrect results.

Here we assume that the environment variable `HOME`, which holds the user's home directory name, has value '/xcssun/users/rms'.

```
(substitute-in-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
```

After substitution, if a '~' or a '/' appears immediately after another '/', the function discards everything before it (up through the immediately preceding '/').

```
(substitute-in-file-name "bar/~/foo")
     ⇒ "~/foo"
(substitute-in-file-name "/usr/local/$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
     ;; '/usr/local/' has been discarded.
```

### 15.8.5 Generating Unique File Names

Some programs need to write temporary files. Here is the usual way to construct a name for such a file:

```
(make-temp-file name-of-application)
```

The job of `make-temp-file` is to prevent two different users or two different jobs from trying to use the exact same file name.

`make-temp-file` *prefix* **&optional** *dir-flag suffix* [Function]

This function creates a temporary file and returns its name. Emacs creates the temporary file's name by adding to *prefix* some random characters that are different in each Emacs job. The result is guaranteed to be a newly created empty file. On MS-DOS, this function can truncate the *string* prefix to fit into the 8+3 file-name limits. If *prefix* is a relative file name, it is expanded against `temporary-file-directory`.

```
(make-temp-file "foo")
     ⇒ "/tmp/foo232J6v"
```

When `make-temp-file` returns, the file has been created and is empty. At that point, you should write the intended contents into the file.

If *dir-flag* is non-`nil`, `make-temp-file` creates an empty directory instead of an empty file. It returns the file name, not the directory name, of that directory. See Section 15.8.3 [Directory Names], page 254.

If *suffix* is non-`nil`, `make-temp-file` adds it at the end of the file name.

To prevent conflicts among different libraries running in the same Emacs, each Lisp program that uses `make-temp-file` should have its own *prefix*. The number added to the end of *prefix* distinguishes between the same application running in different Emacs jobs. Additional added characters permit a large number of distinct names even in one Emacs job.

The default directory for temporary files is controlled by the variable `temporary-file-directory`. This variable gives the user a uniform way to specify the directory for all temporary files. Some programs use `small-temporary-file-directory` instead, if that is non-`nil`. To use it, you should expand the prefix against the proper directory before calling `make-temp-file`.

`temporary-file-directory` [User Option]

This variable specifies the directory name for creating temporary files. Its value should be a directory name (see Section 15.8.3 [Directory Names], page 254), but it is good for Lisp programs to cope if the value is a directory's file name instead. Using the value as the second argument to `expand-file-name` is a good way to achieve that.

The default value is determined in a reasonable way for your operating system; it is based on the `TMPDIR`, `TMP` and `TEMP` environment variables, with a fall-back to a system-dependent name if none of these variables is defined.

Even if you do not use `make-temp-file` to create the temporary file, you should still use this variable to decide which directory to put the file in. However, if you expect the file to be small, you should use `small-temporary-file-directory` first if that is non-`nil`.

`small-temporary-file-directory` [User Option]
This variable specifies the directory name for creating certain temporary files, which are likely to be small.

If you want to write a temporary file which is likely to be small, you should compute the directory like this:

```
(make-temp-file
  (expand-file-name prefix
                    (or small-temporary-file-directory
                        temporary-file-directory)))
```

`make-temp-name` *base-name* [Function]
This function generates a string that can be used as a unique file name. The name starts with *base-name*, and has several random characters appended to it, which are different in each Emacs job. It is like `make-temp-file` except that (i) it just constructs a name, and does not create a file, and (ii) *base-name* should be an absolute file name (on MS-DOS, this function can truncate *base-name* to fit into the 8+3 file-name limits).

**Warning:** In most cases, you should not use this function; use `make-temp-file` instead! This function is susceptible to a race condition, between the `make-temp-name` call and the creation of the file, which in some cases may cause a security hole.

### 15.8.6 File Name Completion

This section describes low-level subroutines for completing a file name. For higher level functions, see ⟨undefined⟩ [Reading File Names], page ⟨undefined⟩.

`file-name-all-completions` *partial-filename directory* [Function]
This function returns a list of all possible completions for a file whose name starts with *partial-filename* in directory *directory*. The order of the completions is the order of the files in the directory, which is unpredictable and conveys no useful information.

The argument *partial-filename* must be a file name containing no directory part and no slash (or backslash on some systems). The current buffer's default directory is prepended to *directory*, if *directory* is not absolute.

In the following example, suppose that '`~rms/lewis`' is the current default directory, and has five files whose names begin with '`f`': '`foo`', '`file~`', '`file.c`', '`file.c.~1~`', and '`file.c.~2~`'.

```
(file-name-all-completions "f" "")
     ⇒ ("foo" "file~" "file.c.~2~"
               "file.c.~1~" "file.c")

(file-name-all-completions "fo" "")
     ⇒ ("foo")
```

`file-name-completion` *filename directory* **&optional** *predicate* [Function]
: This function completes the file name *filename* in directory *directory*. It returns the longest prefix common to all file names in directory *directory* that start with *filename*. If *predicate* is non-`nil` then it ignores possible completions that don't satisfy *predicate*, after calling that function with one argument, the expanded absolute file name.

  If only one match exists and *filename* matches it exactly, the function returns `t`. The function returns `nil` if directory *directory* contains no name starting with *filename*.

  In the following example, suppose that the current default directory has five files whose names begin with '`f`': '`foo`', '`file~`', '`file.c`', '`file.c.~1~`', and '`file.c.~2~`'.

```
(file-name-completion "fi" "")
     ⇒ "file"

(file-name-completion "file.c.~1" "")
     ⇒ "file.c.~1~"

(file-name-completion "file.c.~1~" "")
     ⇒ t

(file-name-completion "file.c.~3" "")
     ⇒ nil
```

`completion-ignored-extensions` [User Option]
: `file-name-completion` usually ignores file names that end in any string in this list. It does not ignore them when all the possible completions end in one of these suffixes. This variable has no effect on `file-name-all-completions`.

  A typical value might look like this:

```
completion-ignored-extensions
     ⇒ (".o" ".elc" "~" ".dvi")
```

  If an element of `completion-ignored-extensions` ends in a slash '`/`', it signals a directory. The elements which do *not* end in a slash will never match a directory; thus, the above value will not filter out a directory named '`foo.elc`'.

### 15.8.7 Standard File Names

Sometimes, an Emacs Lisp program needs to specify a standard file name for a particular use—typically, to hold configuration data specified by the current user. Usually, such files should be located in the directory specified by `user-emacs-directory`, which is '`~/.emacs.d`' by default (see Section 33.4 [Init File], page 711). For example, abbrev definitions are stored by default in '`~/.emacs.d/abbrev_defs`'. The easiest way to specify such a file name is to use the function `locate-user-emacs-file`.

`locate-user-emacs-file` *base-name* **&optional** *old-name* [Function]
: This function returns an absolute file name for an Emacs-specific configuration or data file. The argument '`base-name`' should be a relative file name. The return value is the absolute name of a file in the directory specified by `user-emacs-directory`; if that directory does not exist, this function creates it.

If the optional argument *old-name* is non-`nil`, it specifies a file in the user's home directory, '`~/old-name`'. If such a file exists, the return value is the absolute name of that file, instead of the file specified by *base-name*. This argument is intended to be used by Emacs packages to provide backward compatibility. For instance, prior to the introduction of `user-emacs-directory`, the abbrev file was located in '`~/.abbrev_defs`'. Here is the definition of `abbrev-file-name`:

```
(defcustom abbrev-file-name
  (locate-user-emacs-file "abbrev_defs" ".abbrev_defs")
  "Default name of file from which to read abbrevs."
  ...
  :type 'file)
```

A lower-level function for standardizing file names, which `locate-user-emacs-file` uses as a subroutine, is `convert-standard-filename`.

`convert-standard-filename` *filename* [Function]

This function returns a file name based on *filename*, which fits the conventions of the current operating system.

On GNU and Unix systems, this simply returns *filename*. On other operating systems, it may enforce system-specific file name conventions; for example, on MS-DOS this function performs a variety of changes to enforce MS-DOS file name limitations, including converting any leading '`.`' to '`_`' and truncating to three characters after the '`.`'.

The recommended way to use this function is to specify a name which fits the conventions of GNU and Unix systems, and pass it to `convert-standard-filename`.

## 15.9 Contents of Directories

A directory is a kind of file that contains other files entered under various names. Directories are a feature of the file system.

Emacs can list the names of the files in a directory as a Lisp list, or display the names in a buffer using the `ls` shell command. In the latter case, it can optionally display information about each file, depending on the options passed to the `ls` command.

`directory-files` *directory* **&optional** *full-name match-regexp nosort* [Function]

This function returns a list of the names of the files in the directory *directory*. By default, the list is in alphabetical order.

If *full-name* is non-`nil`, the function returns the files' absolute file names. Otherwise, it returns the names relative to the specified directory.

If *match-regexp* is non-`nil`, this function returns only those file names that contain a match for that regular expression—the other file names are excluded from the list. On case-insensitive filesystems, the regular expression matching is case-insensitive.

If *nosort* is non-`nil`, `directory-files` does not sort the list, so you get the file names in no particular order. Use this if you want the utmost possible speed and don't care what order the files are processed in. If the order of processing is visible to the user, then the user will probably be happier if you do sort the names.

```
(directory-files "~lewis")
     ⇒ ("#foo#" "#foo.el#" "." ".."
         "dired-mods.el" "files.texi"
         "files.texi.~1~")
```

An error is signaled if *directory* is not the name of a directory that can be read.

`directory-files-and-attributes` *directory* **&optional** *full-name match-regexp nosort id-format* [Function]

This is similar to `directory-files` in deciding which files to report on and how to report their names. However, instead of returning a list of file names, it returns for each file a list `(filename . attributes)`, where *attributes* is what `file-attributes` would return for that file. The optional argument *id-format* has the same meaning as the corresponding argument to `file-attributes` (see [Definition of file-attributes], page 245).

`file-expand-wildcards` *pattern* **&optional** *full* [Function]

This function expands the wildcard pattern *pattern*, returning a list of file names that match it.

If *pattern* is written as an absolute file name, the values are absolute also.

If *pattern* is written as a relative file name, it is interpreted relative to the current default directory. The file names returned are normally also relative to the current default directory. However, if *full* is non-`nil`, they are absolute.

`insert-directory` *file switches* **&optional** *wildcard full-directory-p* [Function]

This function inserts (in the current buffer) a directory listing for directory *file*, formatted with `ls` according to *switches*. It leaves point after the inserted text. *switches* may be a string of options, or a list of strings representing individual options.

The argument *file* may be either a directory name or a file specification including wildcard characters. If *wildcard* is non-`nil`, that means treat *file* as a file specification with wildcards.

If *full-directory-p* is non-`nil`, that means the directory listing is expected to show the full contents of a directory. You should specify `t` when *file* is a directory and switches do not contain '`-d`'. (The '`-d`' option to `ls` says to describe a directory itself as a file, rather than showing its contents.)

On most systems, this function works by running a directory listing program whose name is in the variable `insert-directory-program`. If *wildcard* is non-`nil`, it also runs the shell specified by `shell-file-name`, to expand the wildcards.

MS-DOS and MS-Windows systems usually lack the standard Unix program `ls`, so this function emulates the standard Unix program `ls` with Lisp code.

As a technical detail, when *switches* contains the long '`--dired`' option, `insert-directory` treats it specially, for the sake of dired. However, the normally equivalent short '`-D`' option is just passed on to `insert-directory-program`, as any other option.

`insert-directory-program` [Variable]
: This variable's value is the program to run to generate a directory listing for the function `insert-directory`. It is ignored on systems which generate the listing with Lisp code.

## 15.10 Creating, Copying and Deleting Directories

Most Emacs Lisp file-manipulation functions get errors when used on files that are directories. For example, you cannot delete a directory with `delete-file`. These special functions exist to create and delete directories.

`make-directory` *dirname* **&optional** *parents* [Command]
: This command creates a directory named *dirname*. If *parents* is non-`nil`, as is always the case in an interactive call, that means to create the parent directories first, if they don't already exist.

  `mkdir` is an alias for this.

`copy-directory` *dirname newname* **&optional** *keep-time parents* *copy-contents* [Command]
: This command copies the directory named *dirname* to *newname*. If *newname* names an existing directory, *dirname* will be copied to a subdirectory there.

  It always sets the file modes of the copied files to match the corresponding original file.

  The third argument *keep-time* non-`nil` means to preserve the modification time of the copied files. A prefix arg makes *keep-time* non-`nil`.

  The fourth argument *parents* says whether to create parent directories if they don't exist. Interactively, this happens by default.

  The fifth argument *copy-contents*, if non-`nil`, means to copy the contents of *dirname* directly into *newname* if the latter is an existing directory, instead of copying *dirname* into it as a subdirectory.

`delete-directory` *dirname* **&optional** *recursive trash* [Command]
: This command deletes the directory named *dirname*. The function `delete-file` does not work for files that are directories; you must use `delete-directory` for them. If *recursive* is `nil`, and the directory contains any files, `delete-directory` signals an error.

  `delete-directory` only follows symbolic links at the level of parent directories.

  If the optional argument *trash* is non-`nil` and the variable `delete-by-moving-to-trash` is non-`nil`, this command moves the file into the system Trash instead of deleting it. See Section "Miscellaneous File Operations" in *The GNU Emacs Manual*. When called interactively, *trash* is `t` if no prefix argument is given, and `nil` otherwise.

## 15.11 Making Certain File Names "Magic"

You can implement special handling for certain file names. This is called making those names *magic*. The principal use for this feature is in implementing access to remote files (see Section "Remote Files" in *The GNU Emacs Manual*).

To define a kind of magic file name, you must supply a regular expression to define the class of names (all those that match the regular expression), plus a handler that implements all the primitive Emacs file operations for file names that match.

The variable `file-name-handler-alist` holds a list of handlers, together with regular expressions that determine when to apply each handler. Each element has this form:

```
(regexp . handler)
```

All the Emacs primitives for file access and file name transformation check the given file name against `file-name-handler-alist`. If the file name matches *regexp*, the primitives handle that file by calling *handler*.

The first argument given to *handler* is the name of the primitive, as a symbol; the remaining arguments are the arguments that were passed to that primitive. (The first of these arguments is most often the file name itself.) For example, if you do this:

```
(file-exists-p filename)
```

and *filename* has handler *handler*, then *handler* is called like this:

```
(funcall handler 'file-exists-p filename)
```

When a function takes two or more arguments that must be file names, it checks each of those names for a handler. For example, if you do this:

```
(expand-file-name filename dirname)
```

then it checks for a handler for *filename* and then for a handler for *dirname*. In either case, the *handler* is called like this:

```
(funcall handler 'expand-file-name filename dirname)
```

The *handler* then needs to figure out whether to handle *filename* or *dirname*.

If the specified file name matches more than one handler, the one whose match starts last in the file name gets precedence. This rule is chosen so that handlers for jobs such as uncompression are handled first, before handlers for jobs such as remote file access.

Here are the operations that a magic file name handler gets to handle:

`access-file`, `add-name-to-file`,
`byte-compiler-base-file-name`,
`copy-directory`, `copy-file`,
`delete-directory`, `delete-file`,
`diff-latest-backup-file`,
`directory-file-name`,
`directory-files`,
`directory-files-and-attributes`,
`dired-compress-file`, `dired-uncache`,
`expand-file-name`,
`file-accessible-directory-p`,
`file-attributes`,
`file-directory-p`,
`file-executable-p`, `file-exists-p`,
`file-local-copy`, `file-remote-p`,
`file-modes`, `file-name-all-completions`,
`file-name-as-directory`,

`file-name-completion`,
`file-name-directory`,
`file-name-nondirectory`,
`file-name-sans-versions`, `file-newer-than-file-p`,
`file-ownership-preserved-p`,
`file-readable-p`, `file-regular-p`, `file-symlink-p`,
`file-truename`, `file-writable-p`,
`find-backup-file-name`,
`get-file-buffer`,
`insert-directory`,
`insert-file-contents`,
`load`, `make-directory`,
`make-directory-internal`,
`make-symbolic-link`,
`process-file`,
`rename-file`, `set-file-modes`,
`set-visited-file-modtime`, `shell-command`,
`start-file-process`,
`substitute-in-file-name`,
`unhandled-file-name-directory`,
`vc-registered`,
`verify-visited-file-modtime`,
`write-region`.

Handlers for `insert-file-contents` typically need to clear the buffer's modified flag, with `(set-buffer-modified-p nil)`, if the *visit* argument is non-`nil`. This also has the effect of unlocking the buffer if it is locked.

The handler function must handle all of the above operations, and possibly others to be added in the future. It need not implement all these operations itself—when it has nothing special to do for a certain operation, it can reinvoke the primitive, to handle the operation "in the usual way". It should always reinvoke the primitive for an operation it does not recognize. Here's one way to do this:

```
(defun my-file-handler (operation &rest args)
  ;; First check for the specific operations
  ;; that we have special handling for.
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; Handle any operation we don't know about.
        (t (let ((inhibit-file-name-handlers
                  (cons 'my-file-handler
                        (and (eq inhibit-file-name-operation operation)
                             inhibit-file-name-handlers)))
                 (inhibit-file-name-operation operation))
             (apply operation args)))))
```

When a handler function decides to call the ordinary Emacs primitive for the operation at hand, it needs to prevent the primitive from calling the same handler once again, thus leading to an infinite recursion. The example above shows how to do this, with the variables `inhibit-file-name-handlers` and `inhibit-file-name-operation`. Be careful to

use them exactly as shown above; the details are crucial for proper behavior in the case of multiple handlers, and for operations that have two file names that may each have handlers.

Handlers that don't really do anything special for actual access to the file—such as the ones that implement completion of host names for remote file names—should have a non-`nil` `safe-magic` property. For instance, Emacs normally "protects" directory names it finds in `PATH` from becoming magic, if they look like magic file names, by prefixing them with '`/:`'. But if the handler that would be used for them has a non-`nil` `safe-magic` property, the '`/:`' is not added.

A file name handler can have an `operations` property to declare which operations it handles in a nontrivial way. If this property has a non-`nil` value, it should be a list of operations; then only those operations will call the handler. This avoids inefficiency, but its main purpose is for autoloaded handler functions, so that they won't be loaded except when they have real work to do.

Simply deferring all operations to the usual primitives does not work. For instance, if the file name handler applies to `file-exists-p`, then it must handle `load` itself, because the usual `load` code won't work properly in that case. However, if the handler uses the `operations` property to say it doesn't handle `file-exists-p`, then it need not handle `load` nontrivially.

`inhibit-file-name-handlers` [Variable]
: This variable holds a list of handlers whose use is presently inhibited for a certain operation.

`inhibit-file-name-operation` [Variable]
: The operation for which certain handlers are presently inhibited.

`find-file-name-handler` *file operation* [Function]
: This function returns the handler function for file name *file*, or `nil` if there is none. The argument *operation* should be the operation to be performed on the file—the value you will pass to the handler as its first argument when you call it. If *operation* equals `inhibit-file-name-operation`, or if it is not found in the `operations` property of the handler, this function returns `nil`.

`file-local-copy` *filename* [Function]
: This function copies file *filename* to an ordinary non-magic file on the local machine, if it isn't on the local machine already. Magic file names should handle the `file-local-copy` operation if they refer to files on other machines. A magic file name that is used for other purposes than remote file access should not handle `file-local-copy`; then this function will treat the file as local.

  If *filename* is local, whether magic or not, this function does nothing and returns `nil`. Otherwise it returns the file name of the local copy file.

`file-remote-p` *filename* **&optional** *identification connected* [Function]
: This function tests whether *filename* is a remote file. If *filename* is local (not remote), the return value is `nil`. If *filename* is indeed remote, the return value is a string that identifies the remote system.

This identifier string can include a host name and a user name, as well as characters designating the method used to access the remote system. For example, the remote identifier string for the filename `/sudo::/some/file` is `/sudo:root@localhost:`.

If `file-remote-p` returns the same identifier for two different filenames, that means they are stored on the same file system and can be accessed locally with respect to each other. This means, for example, that it is possible to start a remote process accessing both files at the same time. Implementers of file handlers need to ensure this principle is valid.

*identification* specifies which part of the identifier shall be returned as string. *identification* can be the symbol `method`, `user` or `host`; any other value is handled like `nil` and means to return the complete identifier string. In the example above, the remote `user` identifier string would be `root`.

If *connected* is non-`nil`, this function returns `nil` even if *filename* is remote, if Emacs has no network connection to its host. This is useful when you want to avoid the delay of making connections when they don't exist.

`unhandled-file-name-directory` *filename* [Function]

This function returns the name of a directory that is not magic. It uses the directory part of *filename* if that is not magic. For a magic file name, it invokes the file name handler, which therefore decides what value to return. If *filename* is not accessible from a local process, then the file name handler should indicate it by returning `nil`.

This is useful for running a subprocess; every subprocess must have a non-magic directory to serve as its current directory, and this function is a good way to come up with one.

`remote-file-name-inhibit-cache` [User Option]

The attributes of remote files can be cached for better performance. If they are changed outside of Emacs's control, the cached values become invalid, and must be reread.

When this variable is set to `nil`, cached values are never expired. Use this setting with caution, only if you are sure nothing other than Emacs ever changes the remote files. If it is set to `t`, cached values are never used. This is the safest value, but could result in performance degradation.

A compromise is to set it to a positive number. This means that cached values are used for that amount of seconds since they were cached. If a remote file is checked regularly, it might be a good idea to let-bind this variable to a value less than the time period between consecutive checks. For example:

```
(defun display-time-file-nonempty-p (file)
  (let ((remote-file-name-inhibit-cache
         (- display-time-interval 5)))
    (and (file-exists-p file)
         (< 0 (nth 7 (file-attributes
                      (file-chase-links file)))))))
```

## 15.12 File Format Conversion

Emacs performs several steps to convert the data in a buffer (text, text properties, and possibly other information) to and from a representation suitable for storing into a file. This section describes the fundamental functions that perform this *format conversion*, namely `insert-file-contents` for reading a file into a buffer, and `write-region` for writing a buffer into a file.

### 15.12.1 Overview

The function `insert-file-contents`:

- initially, inserts bytes from the file into the buffer;
- decodes bytes to characters as appropriate;
- processes formats as defined by entries in `format-alist`; and
- calls functions in `after-insert-file-functions`.

The function `write-region`:

- initially, calls functions in `write-region-annotate-functions`;
- processes formats as defined by entries in `format-alist`;
- encodes characters to bytes as appropriate; and
- modifies the file with the bytes.

This shows the symmetry of the lowest-level operations; reading and writing handle things in opposite order. The rest of this section describes the two facilities surrounding the three variables named above, as well as some related functions. Section 19.6 [Coding Systems], page 381, for details on character encoding and decoding.

### 15.12.2 Round-Trip Specification

The most general of the two facilities is controlled by the variable `format-alist`, a list of *file format* specifications, which describe textual representations used in files for the data in an Emacs buffer. The descriptions for reading and writing are paired, which is why we call this "round-trip" specification (see Section 15.12.3 [Format Conversion Piecemeal], page 269, for non-paired specification).

`format-alist` [Variable]

This list contains one format definition for each defined file format. Each format definition is a list of this form:

```
(name doc-string regexp from-fn to-fn modify mode-fn preserve)
```

Here is what the elements in a format definition mean:

*name* The name of this format.

*doc-string* A documentation string for the format.

*regexp* A regular expression which is used to recognize files represented in this format. If `nil`, the format is never applied automatically.

*from-fn* A shell command or function to decode data in this format (to convert file data into the usual Emacs data representation).

A shell command is represented as a string; Emacs runs the command as a filter to perform the conversion.

If *from-fn* is a function, it is called with two arguments, *begin* and *end*, which specify the part of the buffer it should convert. It should convert the text by editing it in place. Since this can change the length of the text, *from-fn* should return the modified end position.

One responsibility of *from-fn* is to make sure that the beginning of the file no longer matches *regexp*. Otherwise it is likely to get called again.

*to-fn* A shell command or function to encode data in this format—that is, to convert the usual Emacs data representation into this format.

If *to-fn* is a string, it is a shell command; Emacs runs the command as a filter to perform the conversion.

If *to-fn* is a function, it is called with three arguments: *begin* and *end*, which specify the part of the buffer it should convert, and *buffer*, which specifies which buffer. There are two ways it can do the conversion:

- By editing the buffer in place. In this case, *to-fn* should return the end-position of the range of text, as modified.
- By returning a list of annotations. This is a list of elements of the form `(`*`position . string`*`)`, where *position* is an integer specifying the relative position in the text to be written, and *string* is the annotation to add there. The list must be sorted in order of position when *to-fn* returns it.

  When `write-region` actually writes the text from the buffer to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

*modify* A flag, `t` if the encoding function modifies the buffer, and `nil` if it works by returning a list of annotations.

*mode-fn* A minor-mode function to call after visiting a file converted from this format. The function is called with one argument, the integer 1; that tells a minor-mode function to enable the mode.

*preserve* A flag, `t` if `format-write-file` should not remove this format from `buffer-file-format`.

The function `insert-file-contents` automatically recognizes file formats when it reads the specified file. It checks the text of the beginning of the file against the regular expressions of the format definitions, and if it finds a match, it calls the decoding function for that format. Then it checks all the known formats over again. It keeps checking them until none of them is applicable.

Visiting a file, with `find-file-noselect` or the commands that use it, performs conversion likewise (because it calls `insert-file-contents`); it also calls the mode function for each format that it decodes. It stores a list of the format names in the buffer-local variable `buffer-file-format`.

`buffer-file-format` [Variable]
: This variable states the format of the visited file. More precisely, this is a list of the file format names that were decoded in the course of visiting the current buffer's file. It is always buffer-local in all buffers.

When `write-region` writes data into a file, it first calls the encoding functions for the formats listed in `buffer-file-format`, in the order of appearance in the list.

`format-write-file` *file format* **&optional** *confirm* [Command]
: This command writes the current buffer contents into the file *file* in a format based on *format*, which is a list of format names. It constructs the actual format starting from *format*, then appending any elements from the value of `buffer-file-format` with a non-`nil` *preserve* flag (see above), if they are not already present in *format*. It then updates `buffer-file-format` with this format, making it the default for future saves. Except for the *format* argument, this command is similar to `write-file`. In particular, *confirm* has the same meaning and interactive treatment as the corresponding argument to `write-file`. See [Definition of write-file], page 234.

`format-find-file` *file format* [Command]
: This command finds the file *file*, converting it according to format *format*. It also makes *format* the default if the buffer is saved later.

  The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just RET for *format* specifies `nil`.

`format-insert-file` *file format* **&optional** *beg end* [Command]
: This command inserts the contents of file *file*, converting it according to format *format*. If *beg* and *end* are non-`nil`, they specify which part of the file to read, as in `insert-file-contents` (see Section 15.3 [Reading from Files], page 236).

  The return value is like what `insert-file-contents` returns: a list of the absolute file name and the length of the data inserted (after conversion).

  The argument *format* is a list of format names. If *format* is `nil`, no conversion takes place. Interactively, typing just RET for *format* specifies `nil`.

`buffer-auto-save-file-format` [Variable]
: This variable specifies the format to use for auto-saving. Its value is a list of format names, just like the value of `buffer-file-format`; however, it is used instead of `buffer-file-format` for writing auto-save files. If the value is `t`, the default, auto-saving uses the same format as a regular save in the same buffer. This variable is always buffer-local in all buffers.

### 15.12.3 Piecemeal Specification

In contrast to the round-trip specification described in the previous subsection (see Section 15.12.2 [Format Conversion Round-Trip], page 267), you can use the variables `after-insert-file-functions` and `write-region-annotate-functions` to separately control the respective reading and writing conversions.

Conversion starts with one representation and produces another representation. When there is only one conversion to do, there is no conflict about what to start with. However,

when there are multiple conversions involved, conflict may arise when two conversions need to start with the same data.

This situation is best understood in the context of converting text properties during `write-region`. For example, the character at position 42 in a buffer is '`X`' with a text property `foo`. If the conversion for `foo` is done by inserting into the buffer, say, '`FOO:`', then that changes the character at position 42 from '`X`' to '`F`'. The next conversion will start with the wrong data straight away.

To avoid conflict, cooperative conversions do not modify the buffer, but instead specify *annotations*, a list of elements of the form `(`*`position`*` . `*`string`*`)`, sorted in order of increasing *position*.

If there is more than one conversion, `write-region` merges their annotations destructively into one sorted list. Later, when the text from the buffer is actually written to the file, it intermixes the specified annotations at the corresponding positions. All this takes place without modifying the buffer.

In contrast, when reading, the annotations intermixed with the text are handled immediately. `insert-file-contents` sets point to the beginning of some text to be converted, then calls the conversion functions with the length of that text. These functions should always return with point at the beginning of the inserted text. This approach makes sense for reading because annotations removed by the first converter can't be mistakenly processed by a later converter. Each conversion function should scan for the annotations it recognizes, remove the annotation, modify the buffer text (to set a text property, for example), and return the updated length of the text, as it stands after those changes. The value returned by one function becomes the argument to the next function.

`write-region-annotate-functions` [Variable]

A list of functions for `write-region` to call. Each function in the list is called with two arguments: the start and end of the region to be written. These functions should not alter the contents of the buffer. Instead, they should return annotations.

As a special case, a function may return with a different buffer current. Emacs takes this to mean that the current buffer contains altered text to be output. It therefore changes the *start* and *end* arguments of the `write-region` call, giving them the values of `point-min` and `point-max` in the new buffer, respectively. It also discards all previous annotations, because they should have been dealt with by this function.

`write-region-post-annotation-function` [Variable]

The value of this variable, if non-`nil`, should be a function. This function is called, with no arguments, after `write-region` has completed.

If any function in `write-region-annotate-functions` returns with a different buffer current, Emacs calls `write-region-post-annotation-function` more than once. Emacs calls it with the last buffer that was current, and again with the buffer before that, and so on back to the original buffer.

Thus, a function in `write-region-annotate-functions` can create a buffer, give this variable the local value of `kill-buffer` in that buffer, set up the buffer with altered text, and make the buffer current. The buffer will be killed after `write-region` is done.

`after-insert-file-functions` [Variable]

Each function in this list is called by `insert-file-contents` with one argument, the number of characters inserted, and with point at the beginning of the inserted text. Each function should leave point unchanged, and return the new character count describing the inserted text as modified by the function.

We invite users to write Lisp programs to store and retrieve text properties in files, using these hooks, and thus to experiment with various data formats and find good ones. Eventually we hope users will produce good, general extensions we can install in Emacs.

We suggest not trying to handle arbitrary Lisp objects as text property names or values—because a program that general is probably difficult to write, and slow. Instead, choose a set of possible data types that are reasonably flexible, and not too hard to encode.

# 16 Buffers

A *buffer* is a Lisp object containing text to be edited. Buffers are used to hold the contents of files that are being visited; there may also be buffers that are not visiting files. While several buffers may exist at one time, only one buffer is designated the *current buffer* at any time. Most editing commands act on the contents of the current buffer. Each buffer, including the current buffer, may or may not be displayed in any windows.

## 16.1 Buffer Basics

Buffers in Emacs editing are objects that have distinct names and hold text that can be edited. Buffers appear to Lisp programs as a special data type. You can think of the contents of a buffer as a string that you can extend; insertions and deletions may occur in any part of the buffer. See Chapter 22 [Text], page 454.

A Lisp buffer object contains numerous pieces of information. Some of this information is directly accessible to the programmer through variables, while other information is accessible only through special-purpose functions. For example, the visited file name is directly accessible through a variable, while the value of point is accessible only through a primitive function.

Buffer-specific information that is directly accessible is stored in *buffer-local* variable bindings, which are variable values that are effective only in a particular buffer. This feature allows each buffer to override the values of certain variables. Most major modes override variables such as `fill-column` or `comment-column` in this way. For more information about buffer-local variables and functions related to them, see ⟨undefined⟩ [Buffer-Local Variables], page ⟨undefined⟩.

For functions and variables related to visiting files in buffers, see Section 15.1 [Visiting Files], page 230 and Section 15.2 [Saving Buffers], page 234. For functions and variables related to the display of buffers in windows, see Section 17.10 [Buffers and Windows], page 309.

`bufferp` *object* [Function]

This function returns `t` if *object* is a buffer, `nil` otherwise.

## 16.2 The Current Buffer

There are, in general, many buffers in an Emacs session. At any time, one of them is designated the *current buffer*—the buffer in which most editing takes place. Most of the primitives for examining or changing text operate implicitly on the current buffer (see Chapter 22 [Text], page 454).

Normally, the buffer displayed in the selected window is the current buffer, but this is not always so: a Lisp program can temporarily designate any buffer as current in order to operate on its contents, without changing what is displayed on the screen. The most basic function for designating a current buffer is `set-buffer`.

`current-buffer` [Function]

This function returns the current buffer.

```
(current-buffer)
     ⇒ #<buffer buffers.texi>
```

`set-buffer` *buffer-or-name* [Function]

This function makes *buffer-or-name* the current buffer. *buffer-or-name* must be an existing buffer or the name of an existing buffer. The return value is the buffer made current.

This function does not display the buffer in any window, so the user cannot necessarily see the buffer. But Lisp programs will now operate on it.

When an editing command returns to the editor command loop, Emacs automatically calls `set-buffer` on the buffer shown in the selected window. This is to prevent confusion: it ensures that the buffer that the cursor is in, when Emacs reads a command, is the buffer to which that command applies (see Chapter 2 [Command Loop], page 11). Thus, you should not use `set-buffer` to switch visibly to a different buffer; for that, use the functions described in Section 17.11 [Switching Buffers], page 311.

When writing a Lisp function, do *not* rely on this behavior of the command loop to restore the current buffer after an operation. Editing commands can also be called as Lisp functions by other programs, not just from the command loop; it is convenient for the caller if the subroutine does not change which buffer is current (unless, of course, that is the subroutine's purpose).

To operate temporarily on another buffer, put the `set-buffer` within a `save-current-buffer` form. Here, as an example, is a simplified version of the command `append-to-buffer`:

```
(defun append-to-buffer (buffer start end)
  "Append the text of the region to BUFFER."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-current-buffer
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

Here, we bind a local variable to record the current buffer, and then `save-current-buffer` arranges to make it current again later. Next, `set-buffer` makes the specified buffer current, and `insert-buffer-substring` copies the string from the original buffer to the specified (and now current) buffer.

Alternatively, we can use the `with-current-buffer` macro:

```
(defun append-to-buffer (buffer start end)
  "Append the text of the region to BUFFER."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (with-current-buffer (get-buffer-create buffer)
      (insert-buffer-substring oldbuf start end))))
```

In either case, if the buffer appended to happens to be displayed in some window, the next redisplay will show how its text has changed. If it is not displayed in any window, you will not see the change immediately on the screen. The command causes the buffer to become current temporarily, but does not cause it to be displayed.

If you make local bindings (with `let` or function arguments) for a variable that may also have buffer-local bindings, make sure that the same buffer is current at the beginning

and at the end of the local binding's scope. Otherwise you might bind it in one buffer and unbind it in another!

Do not rely on using `set-buffer` to change the current buffer back, because that won't do the job if a quit happens while the wrong buffer is current. For instance, in the previous example, it would have been wrong to do this:

```
(let ((oldbuf (current-buffer)))
  (set-buffer (get-buffer-create buffer))
  (insert-buffer-substring oldbuf start end)
  (set-buffer oldbuf))
```

Using `save-current-buffer` or `with-current-buffer`, as we did, correctly handles quitting, errors, and `throw`, as well as ordinary evaluation.

`save-current-buffer` *body...* [Special Form]

The `save-current-buffer` special form saves the identity of the current buffer, evaluates the *body* forms, and finally restores that buffer as current. The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see ⟨undefined⟩ [Nonlocal Exits], page ⟨undefined⟩).

If the buffer that used to be current has been killed by the time of exit from `save-current-buffer`, then it is not made current again, of course. Instead, whichever buffer was current just before exit remains current.

`with-current-buffer` *buffer-or-name body...* [Macro]

The `with-current-buffer` macro saves the identity of the current buffer, makes *buffer-or-name* current, evaluates the *body* forms, and finally restores the current buffer. *buffer-or-name* must specify an existing buffer or the name of an existing buffer.

The return value is the value of the last form in *body*. The current buffer is restored even in case of an abnormal exit via `throw` or error (see ⟨undefined⟩ [Nonlocal Exits], page ⟨undefined⟩).

`with-temp-buffer` *body...* [Macro]

The `with-temp-buffer` macro evaluates the *body* forms with a temporary buffer as the current buffer. It saves the identity of the current buffer, creates a temporary buffer and makes it current, evaluates the *body* forms, and finally restores the previous current buffer while killing the temporary buffer. By default, undo information (see Section 22.9 [Undo], page 469) is not recorded in the buffer created by this macro (but *body* can enable that, if needed).

The return value is the value of the last form in *body*. You can return the contents of the temporary buffer by using `(buffer-string)` as the last form.

The current buffer is restored even in case of an abnormal exit via `throw` or error (see ⟨undefined⟩ [Nonlocal Exits], page ⟨undefined⟩).

See also `with-temp-file` in [Writing to Files], page 238.

## 16.3 Buffer Names

Each buffer has a unique name, which is a string. Many of the functions that work on buffers accept either a buffer or a buffer name as an argument. Any argument called *buffer-or-name* is of this sort, and an error is signaled if it is neither a string nor a buffer. Any argument called *buffer* must be an actual buffer object, not a name.

Buffers that are ephemeral and generally uninteresting to the user have names starting with a space, so that the `list-buffers` and `buffer-menu` commands don't mention them (but if such a buffer visits a file, it **is** mentioned). A name starting with space also initially disables recording undo information; see Section 22.9 [Undo], page 469.

`buffer-name` **&optional** *buffer* [Function]

This function returns the name of *buffer* as a string. *buffer* defaults to the current buffer.

If `buffer-name` returns `nil`, it means that *buffer* has been killed. See Section 16.10 [Killing Buffers], page 284.

```
(buffer-name)
     ⇒ "buffers.texi"

(setq foo (get-buffer "temp"))
     ⇒ #<buffer temp>
(kill-buffer foo)
     ⇒ nil
(buffer-name foo)
     ⇒ nil
foo
     ⇒ #<killed buffer>
```

`rename-buffer` *newname* **&optional** *unique* [Command]

This function renames the current buffer to *newname*. An error is signaled if *newname* is not a string.

Ordinarily, `rename-buffer` signals an error if *newname* is already in use. However, if *unique* is non-`nil`, it modifies *newname* to make a name that is not in use. Interactively, you can make *unique* non-`nil` with a numeric prefix argument. (This is how the command `rename-uniquely` is implemented.)

This function returns the name actually given to the buffer.

`get-buffer` *buffer-or-name* [Function]

This function returns the buffer specified by *buffer-or-name*. If *buffer-or-name* is a string and there is no buffer with that name, the value is `nil`. If *buffer-or-name* is a buffer, it is returned as given; that is not very useful, so the argument is usually a name. For example:

```
(setq b (get-buffer "lewis"))
     ⇒ #<buffer lewis>
(get-buffer b)
     ⇒ #<buffer lewis>
(get-buffer "Frazzle-nots")
     ⇒ nil
```

See also the function `get-buffer-create` in Section 16.9 [Creating Buffers], page 284.

`generate-new-buffer-name` *starting-name* **&optional** *ignore* [Function]
This function returns a name that would be unique for a new buffer—but does not create the buffer. It starts with *starting-name*, and produces a name not currently in use for any buffer by appending a number inside of '`<...>`'. It starts at 2 and keeps incrementing the number until it is not the name of an existing buffer.

If the optional second argument *ignore* is non-`nil`, it should be a string, a potential buffer name. It means to consider that potential buffer acceptable, if it is tried, even it is the name of an existing buffer (which would normally be rejected). Thus, if buffers named '`foo`', '`foo<2>`', '`foo<3>`' and '`foo<4>`' exist,

```
(generate-new-buffer-name "foo")
     ⇒ "foo<5>"
(generate-new-buffer-name "foo" "foo<3>")
     ⇒ "foo<3>"
(generate-new-buffer-name "foo" "foo<6>")
     ⇒ "foo<5>"
```

See the related function `generate-new-buffer` in Section 16.9 [Creating Buffers], page 284.

## 16.4 Buffer File Name

The *buffer file name* is the name of the file that is visited in that buffer. When a buffer is not visiting a file, its buffer file name is `nil`. Most of the time, the buffer name is the same as the nondirectory part of the buffer file name, but the buffer file name and the buffer name are distinct and can be set independently. See Section 15.1 [Visiting Files], page 230.

`buffer-file-name` **&optional** *buffer* [Function]
This function returns the absolute file name of the file that *buffer* is visiting. If *buffer* is not visiting any file, `buffer-file-name` returns `nil`. If *buffer* is not supplied, it defaults to the current buffer.

```
(buffer-file-name (other-buffer))
     ⇒ "/usr/user/lewis/manual/files.texi"
```

`buffer-file-name` [Variable]
This buffer-local variable contains the name of the file being visited in the current buffer, or `nil` if it is not visiting a file. It is a permanent local variable, unaffected by `kill-all-local-variables`.

```
buffer-file-name
     ⇒ "/usr/user/lewis/manual/buffers.texi"
```

It is risky to change this variable's value without doing various other things. Normally it is better to use `set-visited-file-name` (see below); some of the things done there, such as changing the buffer name, are not strictly necessary, but others are essential to avoid confusing Emacs.

`buffer-file-truename` [Variable]
This buffer-local variable holds the abbreviated truename of the file visited in the current buffer, or `nil` if no file is visited. It is a permanent local, unaffected by `kill-`

`all-local-variables`. See Section 15.6.3 [Truenames], page 243, and [abbreviate-file-name], page 255.

`buffer-file-number` [Variable]
This buffer-local variable holds the file number and directory device number of the file visited in the current buffer, or `nil` if no file or a nonexistent file is visited. It is a permanent local, unaffected by `kill-all-local-variables`.

The value is normally a list of the form `(filenum devnum)`. This pair of numbers uniquely identifies the file among all files accessible on the system. See the function `file-attributes`, in Section 15.6.4 [File Attributes], page 244, for more information about them.

If `buffer-file-name` is the name of a symbolic link, then both numbers refer to the recursive target.

`get-file-buffer` *filename* [Function]
This function returns the buffer visiting file *filename*. If there is no such buffer, it returns `nil`. The argument *filename*, which must be a string, is expanded (see Section 15.8.4 [File Name Expansion], page 255), then compared against the visited file names of all live buffers. Note that the buffer's `buffer-file-name` must match the expansion of *filename* exactly. This function will not recognize other names for the same file.

```
(get-file-buffer "buffers.texi")
    ⇒ #<buffer buffers.texi>
```

In unusual circumstances, there can be more than one buffer visiting the same file name. In such cases, this function returns the first such buffer in the buffer list.

`find-buffer-visiting` *filename* **&optional** *predicate* [Function]
This is like `get-file-buffer`, except that it can return any buffer visiting the file *possibly under a different name*. That is, the buffer's `buffer-file-name` does not need to match the expansion of *filename* exactly, it only needs to refer to the same file. If *predicate* is non-`nil`, it should be a function of one argument, a buffer visiting *filename*. The buffer is only considered a suitable return value if *predicate* returns non-`nil`. If it can not find a suitable buffer to return, `find-buffer-visiting` returns `nil`.

`set-visited-file-name` *filename* **&optional** *no-query along-with-file* [Command]
If *filename* is a non-empty string, this function changes the name of the file visited in the current buffer to *filename*. (If the buffer had no visited file, this gives it one.) The *next time* the buffer is saved it will go in the newly-specified file.

This command marks the buffer as modified, since it does not (as far as Emacs knows) match the contents of *filename*, even if it matched the former visited file. It also renames the buffer to correspond to the new file name, unless the new name is already in use.

If *filename* is `nil` or the empty string, that stands for "no visited file". In this case, `set-visited-file-name` marks the buffer as having no visited file, without changing the buffer's modified flag.

Normally, this function asks the user for confirmation if there already is a buffer visiting *filename*. If *no-query* is non-`nil`, that prevents asking this question. If there already is a buffer visiting *filename*, and the user confirms or *query* is non-`nil`, this function makes the new buffer name unique by appending a number inside of '`<...>`' to *filename*.

If *along-with-file* is non-`nil`, that means to assume that the former visited file has been renamed to *filename*. In this case, the command does not change the buffer's modified flag, nor the buffer's recorded last file modification time as reported by `visited-file-modtime` (see Section 16.6 [Modification Time], page 279). If *along-with-file* is `nil`, this function clears the recorded last file modification time, after which `visited-file-modtime` returns zero.

When the function `set-visited-file-name` is called interactively, it prompts for *filename* in the minibuffer.

`list-buffers-directory` [Variable]

This buffer-local variable specifies a string to display in a buffer listing where the visited file name would go, for buffers that don't have a visited file name. Dired buffers use this variable.

## 16.5 Buffer Modification

Emacs keeps a flag called the *modified flag* for each buffer, to record whether you have changed the text of the buffer. This flag is set to `t` whenever you alter the contents of the buffer, and cleared to `nil` when you save it. Thus, the flag shows whether there are unsaved changes. The flag value is normally shown in the mode line (see Section 20.4.4 [Mode Line Variables], page 422), and controls saving (see Section 15.2 [Saving Buffers], page 234) and auto-saving (see ⟨undefined⟩ [Auto-Saving], page ⟨undefined⟩).

Some Lisp programs set the flag explicitly. For example, the function `set-visited-file-name` sets the flag to `t`, because the text does not match the newly-visited file, even if it is unchanged from the file formerly visited.

The functions that modify the contents of buffers are described in Chapter 22 [Text], page 454.

`buffer-modified-p` **&optional** *buffer* [Function]

This function returns `t` if the buffer *buffer* has been modified since it was last read in from a file or saved, or `nil` otherwise. If *buffer* is not supplied, the current buffer is tested.

`set-buffer-modified-p` *flag* [Function]

This function marks the current buffer as modified if *flag* is non-`nil`, or as unmodified if the flag is `nil`.

Another effect of calling this function is to cause unconditional redisplay of the mode line for the current buffer. In fact, the function `force-mode-line-update` works by doing this:

```
(set-buffer-modified-p (buffer-modified-p))
```

`restore-buffer-modified-p` *flag* [Function]

Like `set-buffer-modified-p`, but does not force redisplay of mode lines.

`not-modified` **&optional** *arg* [Command]
: This command marks the current buffer as unmodified, and not needing to be saved. If *arg* is non-`nil`, it marks the buffer as modified, so that it will be saved at the next suitable occasion. Interactively, *arg* is the prefix argument.

  Don't use this function in programs, since it prints a message in the echo area; use `set-buffer-modified-p` (above) instead.

`buffer-modified-tick` **&optional** *buffer* [Function]
: This function returns *buffer*'s modification-count. This is a counter that increments every time the buffer is modified. If *buffer* is `nil` (or omitted), the current buffer is used. The counter can wrap around occasionally.

`buffer-chars-modified-tick` **&optional** *buffer* [Function]
: This function returns *buffer*'s character-change modification-count. Changes to text properties leave this counter unchanged; however, each time text is inserted or removed from the buffer, the counter is reset to the value that would be returned by `buffer-modified-tick`. By comparing the values returned by two `buffer-chars-modified-tick` calls, you can tell whether a character change occurred in that buffer in between the calls. If *buffer* is `nil` (or omitted), the current buffer is used.

## 16.6 Buffer Modification Time

Suppose that you visit a file and make changes in its buffer, and meanwhile the file itself is changed on disk. At this point, saving the buffer would overwrite the changes in the file. Occasionally this may be what you want, but usually it would lose valuable information. Emacs therefore checks the file's modification time using the functions described below before saving the file. (See Section 15.6.4 [File Attributes], page 244, for how to examine a file's modification time.)

`verify-visited-file-modtime` **&optional** *buffer* [Function]
: This function compares what *buffer* (by default, the current-buffer) has recorded for the modification time of its visited file against the actual modification time of the file as recorded by the operating system. The two should be the same unless some other process has written the file since Emacs visited or saved it.

  The function returns `t` if the last actual modification time and Emacs's recorded modification time are the same, `nil` otherwise. It also returns `t` if the buffer has no recorded last modification time, that is if `visited-file-modtime` would return zero.

  It always returns `t` for buffers that are not visiting a file, even if `visited-file-modtime` returns a non-zero value. For instance, it always returns `t` for dired buffers. It returns `t` for buffers that are visiting a file that does not exist and never existed, but `nil` for file-visiting buffers whose file has been deleted.

`clear-visited-file-modtime` [Function]
: This function clears out the record of the last modification time of the file being visited by the current buffer. As a result, the next attempt to save this buffer will not complain of a discrepancy in file modification times.

  This function is called in `set-visited-file-name` and other exceptional places where the usual test to avoid overwriting a changed file should not be done.

`visited-file-modtime` [Function]

This function returns the current buffer's recorded last file modification time, as a list of the form `(high low microsec picosec)`. (This is the same format that `file-attributes` uses to return time values; see Section 15.6.4 [File Attributes], page 244.)

If the buffer has no recorded last modification time, this function returns zero. This case occurs, for instance, if the buffer is not visiting a file or if the time has been explicitly cleared by `clear-visited-file-modtime`. Note, however, that `visited-file-modtime` returns a list for some non-file buffers too. For instance, in a Dired buffer listing a directory, it returns the last modification time of that directory, as recorded by Dired.

For a new buffer visiting a not yet existing file, *high* is −1 and *low* is 65535, that is, $2^{16} - 1$.

`set-visited-file-modtime` **&optional** *time* [Function]

This function updates the buffer's record of the last modification time of the visited file, to the value specified by *time* if *time* is not `nil`, and otherwise to the last modification time of the visited file.

If *time* is neither `nil` nor zero, it should have the form `(high low microsec picosec)`, the format used by `current-time` (see ⟨undefined⟩ [Time of Day], page ⟨undefined⟩).

This function is useful if the buffer was not read from the file normally, or if the file itself has been changed for some known benign reason.

`ask-user-about-supersession-threat` *filename* [Function]

This function is used to ask a user how to proceed after an attempt to modify an buffer visiting file *filename* when the file is newer than the buffer text. Emacs detects this because the modification time of the file on disk is newer than the last save-time of the buffer. This means some other program has probably altered the file.

Depending on the user's answer, the function may return normally, in which case the modification of the buffer proceeds, or it may signal a `file-supersession` error with data `(filename)`, in which case the proposed buffer modification is not allowed.

This function is called automatically by Emacs on the proper occasions. It exists so you can customize Emacs by redefining it. See the file '`userlock.el`' for the standard definition.

See also the file locking mechanism in Section 15.5 [File Locks], page 239.

## 16.7 Read-Only Buffers

If a buffer is *read-only*, then you cannot change its contents, although you may change your view of the contents by scrolling and narrowing.

Read-only buffers are used in two kinds of situations:

- A buffer visiting a write-protected file is normally read-only.

  Here, the purpose is to inform the user that editing the buffer with the aim of saving it in the file may be futile or undesirable. The user who wants to change the buffer text despite this can do so after clearing the read-only flag with `C-x C-q`.

- Modes such as Dired and Rmail make buffers read-only when altering the contents with the usual editing commands would probably be a mistake.

  The special commands of these modes bind `buffer-read-only` to `nil` (with `let`) or bind `inhibit-read-only` to `t` around the places where they themselves change the text.

`buffer-read-only` [Variable]
: This buffer-local variable specifies whether the buffer is read-only. The buffer is read-only if this variable is non-`nil`.

`inhibit-read-only` [Variable]
: If this variable is non-`nil`, then read-only buffers and, depending on the actual value, some or all read-only characters may be modified. Read-only characters in a buffer are those that have a non-`nil` `read-only` text property. See Section 22.19.4 [Special Properties], page 494, for more information about text properties.

  If `inhibit-read-only` is `t`, all `read-only` character properties have no effect. If `inhibit-read-only` is a list, then `read-only` character properties have no effect if they are members of the list (comparison is done with `eq`).

`read-only-mode` **&optional** *arg* [Command]
: This is the mode command for Read Only minor mode, a buffer-local minor mode. When the mode is enabled, `buffer-read-only` is non-`nil` in the buffer; when disabled, `buffer-read-only` is `nil` in the buffer. The calling convention is the same as for other minor mode commands (see Section 20.3.1 [Minor Mode Conventions], page 413).

  This minor mode mainly serves as a wrapper for `buffer-read-only`; unlike most minor modes, there is no separate `read-only-mode` variable. Even when Read Only mode is disabled, characters with non-`nil` `read-only` text properties remain read-only. To temporarily ignore all read-only states, bind `inhibit-read-only`, as described above.

  When enabling Read Only mode, this mode command also enables View mode if the option `view-read-only` is non-`nil`. See Section "Miscellaneous Buffer Operations" in *The GNU Emacs Manual*. When disabling Read Only mode, it disables View mode if View mode was enabled.

`barf-if-buffer-read-only` [Function]
: This function signals a `buffer-read-only` error if the current buffer is read-only. See Section 2.2.1 [Using Interactive], page 12, for another way to signal an error if the current buffer is read-only.

## 16.8 The Buffer List

The *buffer list* is a list of all live buffers. The order of the buffers in this list is based primarily on how recently each buffer has been displayed in a window. Several functions, notably `other-buffer`, use this ordering. A buffer list displayed for the user also follows this order.

Creating a buffer adds it to the end of the buffer list, and killing a buffer removes it from that list. A buffer moves to the front of this list whenever it is chosen for display

in a window (see Section 17.11 [Switching Buffers], page 311) or a window displaying it is selected (see Section 17.8 [Selecting Windows], page 306). A buffer moves to the end of the list when it is buried (see `bury-buffer`, below). There are no functions available to the Lisp programmer which directly manipulate the buffer list.

In addition to the fundamental buffer list just described, Emacs maintains a local buffer list for each frame, in which the buffers that have been displayed (or had their windows selected) in that frame come first. (This order is recorded in the frame's `buffer-list` frame parameter; see Section 18.3.3.5 [Buffer Parameters], page 350.) Buffers never displayed in that frame come afterward, ordered according to the fundamental buffer list.

`buffer-list` **&optional** *frame* [Function]

This function returns the buffer list, including all buffers, even those whose names begin with a space. The elements are actual buffers, not their names.

If *frame* is a frame, this returns *frame*'s local buffer list. If *frame* is `nil` or omitted, the fundamental buffer list is used: the buffers appear in order of most recent display or selection, regardless of which frames they were displayed on.

```
(buffer-list)
     ⇒ (#<buffer buffers.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)

;; Note that the name of the minibuffer
;;    begins with a space!
(mapcar (function buffer-name) (buffer-list))
    ⇒ ("buffers.texi" " *Minibuf-1*"
        "buffer.c" "*Help*" "TAGS")
```

The list returned by `buffer-list` is constructed specifically; it is not an internal Emacs data structure, and modifying it has no effect on the order of buffers. If you want to change the order of buffers in the fundamental buffer list, here is an easy way:

```
(defun reorder-buffer-list (new-list)
  (while new-list
    (bury-buffer (car new-list))
    (setq new-list (cdr new-list))))
```

With this method, you can specify any order for the list, but there is no danger of losing a buffer or adding something that is not a valid live buffer.

To change the order or value of a specific frame's buffer list, set that frame's `buffer-list` parameter with `modify-frame-parameters` (see Section 18.3.1 [Parameter Access], page 345).

`other-buffer` **&optional** *buffer visible-ok frame* [Function]

This function returns the first buffer in the buffer list other than *buffer*. Usually, this is the buffer appearing in the most recently selected window (in frame *frame* or else the selected frame, see Section 18.9 [Input Focus], page 358), aside from *buffer*. Buffers whose names start with a space are not considered at all.

If *buffer* is not supplied (or if it is not a live buffer), then `other-buffer` returns the first buffer in the selected frame's local buffer list. (If *frame* is non-`nil`, it returns the first buffer in *frame*'s local buffer list instead.)

If *frame* has a non-`nil` `buffer-predicate` parameter, then `other-buffer` uses that predicate to decide which buffers to consider. It calls the predicate once for each buffer, and if the value is `nil`, that buffer is ignored. See Section 18.3.3.5 [Buffer Parameters], page 350.

If *visible-ok* is `nil`, `other-buffer` avoids returning a buffer visible in any window on any visible frame, except as a last resort. If *visible-ok* is non-`nil`, then it does not matter whether a buffer is displayed somewhere or not.

If no suitable buffer exists, the buffer '`*scratch*`' is returned (and created, if necessary).

`last-buffer` **&optional** *buffer visible-ok frame* [Function]

This function returns the last buffer in *frame*'s buffer list other than *BUFFER*. If *frame* is omitted or `nil`, it uses the selected frame's buffer list.

The argument *visible-ok* is handled as with `other-buffer`, see above. If no suitable buffer can be found, the buffer '`*scratch*`' is returned.

`bury-buffer` **&optional** *buffer-or-name* [Command]

This command puts *buffer-or-name* at the end of the buffer list, without changing the order of any of the other buffers on the list. This buffer therefore becomes the least desirable candidate for `other-buffer` to return. The argument can be either a buffer itself or the name of one.

This function operates on each frame's `buffer-list` parameter as well as the fundamental buffer list; therefore, the buffer that you bury will come last in the value of `(buffer-list` *`frame`*`)` and in the value of `(buffer-list)`. In addition, it also puts the buffer at the end of the list of buffer of the selected window (see Section 17.15 [Window History], page 319) provided it is shown in that window.

If *buffer-or-name* is `nil` or omitted, this means to bury the current buffer. In addition, if the current buffer is displayed in the selected window, this makes sure that the window is either deleted or another buffer is shown in it. More precisely, if the selected window is dedicated (see Section 17.16 [Dedicated Windows], page 321) and there are other windows on its frame, the window is deleted. If it is the only window on its frame and that frame is not the only frame on its terminal, the frame is "dismissed" by calling the function specified by `frame-auto-hide-function` (see Section 17.17 [Quitting Windows], page 321). Otherwise, it calls `switch-to-prev-buffer` (see Section 17.15 [Window History], page 319) to show another buffer in that window. If *buffer-or-name* is displayed in some other window, it remains displayed there.

To replace a buffer in all the windows that display it, use `replace-buffer-in-windows`, See Section 17.10 [Buffers and Windows], page 309.

`unbury-buffer` [Command]

This command switches to the last buffer in the local buffer list of the selected frame. More precisely, it calls the function `switch-to-buffer` (see Section 17.11 [Switching Buffers], page 311), to display the buffer returned by `last-buffer` (see above), in the selected window.

## 16.9 Creating Buffers

This section describes the two primitives for creating buffers. `get-buffer-create` creates a buffer if it finds no existing buffer with the specified name; `generate-new-buffer` always creates a new buffer and gives it a unique name.

Other functions you can use to create buffers include `with-output-to-temp-buffer` (see Section 11.8 [Temporary Displays], page 125) and `create-file-buffer` (see Section 15.1 [Visiting Files], page 230). Starting a subprocess can also create a buffer (see ⟨undefined⟩ [Processes], page ⟨undefined⟩).

`get-buffer-create` *buffer-or-name* [Function]

This function returns a buffer named *buffer-or-name*. The buffer returned does not become the current buffer—this function does not change which buffer is current.

*buffer-or-name* must be either a string or an existing buffer. If it is a string and a live buffer with that name already exists, `get-buffer-create` returns that buffer. If no such buffer exists, it creates a new buffer. If *buffer-or-name* is a buffer instead of a string, it is returned as given, even if it is dead.

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

The major mode for a newly created buffer is set to Fundamental mode. (The default value of the variable `major-mode` is handled at a higher level; see Section 20.2.2 [Auto Major Mode], page 403.) If the name begins with a space, the buffer initially disables undo information recording (see Section 22.9 [Undo], page 469).

`generate-new-buffer` *name* [Function]

This function returns a newly created, empty buffer, but does not make it current. The name of the buffer is generated by passing *name* to the function `generate-new-buffer-name` (see Section 16.3 [Buffer Names], page 275). Thus, if there is no buffer named *name*, then that is the name of the new buffer; if that name is in use, a suffix of the form '`<n>`', where *n* is an integer, is appended to *name*.

An error is signaled if *name* is not a string.

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

The major mode for the new buffer is set to Fundamental mode. The default value of the variable `major-mode` is handled at a higher level. See Section 20.2.2 [Auto Major Mode], page 403.

## 16.10 Killing Buffers

*Killing a buffer* makes its name unknown to Emacs and makes the memory space it occupied available for other use.

The buffer object for the buffer that has been killed remains in existence as long as anything refers to it, but it is specially marked so that you cannot make it current or

display it. Killed buffers retain their identity, however; if you kill two distinct buffers, they remain distinct according to `eq` although both are dead.

If you kill a buffer that is current or displayed in a window, Emacs automatically selects or displays some other buffer instead. This means that killing a buffer can change the current buffer. Therefore, when you kill a buffer, you should also take the precautions associated with changing the current buffer (unless you happen to know that the buffer being killed isn't current). See Section 16.2 [Current Buffer], page 272.

If you kill a buffer that is the base buffer of one or more indirect buffers, the indirect buffers are automatically killed as well.

The `buffer-name` of a buffer is `nil` if, and only if, the buffer is killed. A buffer that has not been killed is called a *live* buffer. To test whether a buffer is live or killed, use the function `buffer-live-p` (see below).

`kill-buffer` **&optional** *buffer-or-name* [Command]

This function kills the buffer *buffer-or-name*, freeing all its memory for other uses or to be returned to the operating system. If *buffer-or-name* is `nil` or omitted, it kills the current buffer.

Any processes that have this buffer as the `process-buffer` are sent the `SIGHUP` ("hangup") signal, which normally causes them to terminate. See ⟨undefined⟩ [Signals to Processes], page ⟨undefined⟩.

If the buffer is visiting a file and contains unsaved changes, `kill-buffer` asks the user to confirm before the buffer is killed. It does this even if not called interactively. To prevent the request for confirmation, clear the modified flag before calling `kill-buffer`. See Section 16.5 [Buffer Modification], page 278.

This function calls `replace-buffer-in-windows` for cleaning up all windows currently displaying the buffer to be killed.

Killing a buffer that is already dead has no effect.

This function returns `t` if it actually killed the buffer. It returns `nil` if the user refuses to confirm or if *buffer-or-name* was already dead.

```
(kill-buffer "foo.unchanged")
     ⇒ t
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     ⇒ t
```

`kill-buffer-query-functions` [Variable]

After confirming unsaved changes, `kill-buffer` calls the functions in the list `kill-buffer-query-functions`, in order of appearance, with no arguments. The buffer being killed is the current buffer when they are called. The idea of this feature is that these functions will ask for confirmation from the user. If any of them returns `nil`, `kill-buffer` spares the buffer's life.

`kill-buffer-hook` [Variable]

This is a normal hook run by `kill-buffer` after asking all the questions it is going to ask, just before actually killing the buffer. The buffer to be killed is current when

the hook functions run. See Section 33.2.2 [Hooks], page 696. This variable is a permanent local, so its local binding is not cleared by changing major modes.

`buffer-offer-save` [User Option]
This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` (if the second optional argument to that function is `t`) to offer to save that buffer, just as they offer to save file-visiting buffers. See [Definition of save-some-buffers], page 234. The variable `buffer-offer-save` automatically becomes buffer-local when set for any reason. See ⟨undefined⟩ [Buffer-Local Variables], page ⟨undefined⟩.

`buffer-save-without-query` [Variable]
This variable, if non-`nil` in a particular buffer, tells `save-buffers-kill-emacs` and `save-some-buffers` to save this buffer (if it's modified) without asking the user. The variable automatically becomes buffer-local when set for any reason.

`buffer-live-p` *object* [Function]
This function returns `t` if *object* is a live buffer (a buffer which has not been killed), `nil` otherwise.

## 16.11 Indirect Buffers

An *indirect buffer* shares the text of some other buffer, which is called the *base buffer* of the indirect buffer. In some ways it is the analogue, for buffers, of a symbolic link among files. The base buffer may not itself be an indirect buffer.

The text of the indirect buffer is always identical to the text of its base buffer; changes made by editing either one are visible immediately in the other. This includes the text properties as well as the characters themselves.

In all other respects, the indirect buffer and its base buffer are completely separate. They have different names, independent values of point, independent narrowing, independent markers and overlays (though inserting or deleting text in either buffer relocates the markers and overlays for both), independent major modes, and independent buffer-local variable bindings.

An indirect buffer cannot visit a file, but its base buffer can. If you try to save the indirect buffer, that actually saves the base buffer.

Killing an indirect buffer has no effect on its base buffer. Killing the base buffer effectively kills the indirect buffer in that it cannot ever again be the current buffer.

`make-indirect-buffer` *base-buffer name* **&optional** *clone* [Command]
This creates and returns an indirect buffer named *name* whose base buffer is *base-buffer*. The argument *base-buffer* may be a live buffer or the name (a string) of an existing buffer. If *name* is the name of an existing buffer, an error is signaled.

If *clone* is non-`nil`, then the indirect buffer originally shares the "state" of *base-buffer* such as major mode, minor modes, buffer local variables and so on. If *clone* is omitted or `nil` the indirect buffer's state is set to the default state for new buffers.

If *base-buffer* is an indirect buffer, its base buffer is used as the base for the new buffer. If, in addition, *clone* is non-`nil`, the initial state is copied from the actual base buffer, not from *base-buffer*.

`clone-indirect-buffer` *newname display-flag* **&optional** *norecord* [Command]
: This function creates and returns a new indirect buffer that shares the current buffer's base buffer and copies the rest of the current buffer's attributes. (If the current buffer is not indirect, it is used as the base buffer.)

  If *display-flag* is non-`nil`, that means to display the new buffer by calling `pop-to-buffer`. If *norecord* is non-`nil`, that means not to put the new buffer to the front of the buffer list.

`buffer-base-buffer` **&optional** *buffer* [Function]
: This function returns the base buffer of *buffer*, which defaults to the current buffer. If *buffer* is not indirect, the value is `nil`. Otherwise, the value is another buffer, which is never an indirect buffer.

## 16.12 Swapping Text Between Two Buffers

Specialized modes sometimes need to let the user access from the same buffer several vastly different types of text. For example, you may need to display a summary of the buffer text, in addition to letting the user access the text itself.

This could be implemented with multiple buffers (kept in sync when the user edits the text), or with narrowing (see ⟨undefined⟩ [Narrowing], page ⟨undefined⟩). But these alternatives might sometimes become tedious or prohibitively expensive, especially if each type of text requires expensive buffer-global operations in order to provide correct display and editing commands.

Emacs provides another facility for such modes: you can quickly swap buffer text between two buffers with `buffer-swap-text`. This function is very fast because it doesn't move any text, it only changes the internal data structures of the buffer object to point to a different chunk of text. Using it, you can pretend that a group of two or more buffers are actually a single virtual buffer that holds the contents of all the individual buffers together.

`buffer-swap-text` *buffer* [Function]
: This function swaps the text of the current buffer and that of its argument *buffer*. It signals an error if one of the two buffers is an indirect buffer (see Section 16.11 [Indirect Buffers], page 286) or is a base buffer of an indirect buffer.

  All the buffer properties that are related to the buffer text are swapped as well: the positions of point and mark, all the markers, the overlays, the text properties, the undo list, the value of the `enable-multibyte-characters` flag (see ⟨undefined⟩ [Text Representations], page ⟨undefined⟩), etc.

If you use `buffer-swap-text` on a file-visiting buffer, you should set up a hook to save the buffer's original text rather than what it was swapped with. `write-region-annotate-functions` works for this purpose. You should probably set `buffer-saved-size` to $-2$ in the buffer, so that changes in the text it is swapped with will not interfere with auto-saving.

## 16.13 The Buffer Gap

Emacs buffers are implemented using an invisible *gap* to make insertion and deletion faster. Insertion works by filling in part of the gap, and deletion adds to the gap. Of course, this means that the gap must first be moved to the locus of the insertion or deletion. Emacs

moves the gap only when you try to insert or delete. This is why your first editing command in one part of a large buffer, after previously editing in another far-away part, sometimes involves a noticeable delay.

This mechanism works invisibly, and Lisp code should never be affected by the gap's current location, but these functions are available for getting information about the gap status.

`gap-position` [Function]
: This function returns the current gap position in the current buffer.

`gap-size` [Function]
: This function returns the current gap size of the current buffer.

# 17 Windows

This chapter describes the functions and variables related to Emacs windows. See Chapter 18 [Frames], page 341, for how windows are assigned an area of screen available for Emacs to use. See Chapter 11 [Display], page 111, for information on how text is displayed in windows.

## 17.1 Basic Concepts of Emacs Windows

A *window* is an area of the screen that is used to display a buffer (see Chapter 16 [Buffers], page 272). In Emacs Lisp, windows are represented by a special Lisp object type.

Windows are grouped into frames (see Chapter 18 [Frames], page 341). Each frame contains at least one window; the user can subdivide it into multiple, non-overlapping windows to view several buffers at once. Lisp programs can use multiple windows for a variety of purposes. In Rmail, for example, you can view a summary of message titles in one window, and the contents of the selected message in another window.

Emacs uses the word "window" with a different meaning than in graphical desktop environments and window systems, such as the X Window System. When Emacs is run on X, each of its graphical X windows is an Emacs frame (containing one or more Emacs windows). When Emacs is run on a text terminal, the frame fills the entire terminal screen.

Unlike X windows, Emacs windows are *tiled*; they never overlap within the area of the frame. When a window is created, resized, or deleted, the change in window space is taken from or given to the adjacent windows, so that the total area of the frame is unchanged.

`windowp` *object* [Function]
: This function returns `t` if *object* is a window (whether or not it displays a buffer). Otherwise, it returns `nil`.

A *live window* is one that is actually displaying a buffer in a frame.

`window-live-p` *object* [Function]
: This function returns `t` if *object* is a live window and `nil` otherwise. A live window is one that displays a buffer.

The windows in each frame are organized into a *window tree*. See Section 17.2 [Windows and Frames], page 290. The leaf nodes of each window tree are live windows—the ones actually displaying buffers. The internal nodes of the window tree are *internal windows*, which are not live.

A *valid window* is one that is either live or internal. A valid window can be *deleted*, i.e., removed from its frame (see Section 17.6 [Deleting Windows], page 300); then it is no longer valid, but the Lisp object representing it might be still referenced from other Lisp objects. A deleted window may be made valid again by restoring a saved window configuration (see Section 17.24 [Window Configurations], page 335).

You can distinguish valid windows from deleted windows with `window-valid-p`.

`window-valid-p` *object* [Function]
: This function returns `t` if *object* is a live window, or an internal window in a window tree. Otherwise, it returns `nil`, including for the case where *object* is a deleted window.

In each frame, at any time, exactly one Emacs window is designated as *selected within the frame*. For the selected frame, that window is called the *selected window*—the one in which most editing takes place, and in which the cursor for selected windows appears (see Section 18.3.3.7 [Cursor Parameters], page 351). The selected window's buffer is usually also the current buffer, except when `set-buffer` has been used (see Section 16.2 [Current Buffer], page 272). As for non-selected frames, the window selected within the frame becomes the selected window if the frame is ever selected. See Section 17.8 [Selecting Windows], page 306.

`selected-window` [Function]
: This function returns the selected window (which is always a live window).

## 17.2 Windows and Frames

Each window belongs to exactly one frame (see Chapter 18 [Frames], page 341).

`window-frame` *window* [Function]
: This function returns the frame that the window *window* belongs to. If *window* is `nil`, it defaults to the selected window.

`window-list` **&optional** *frame minibuffer window* [Function]
: This function returns a list of live windows belonging to the frame *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame.

  The optional argument *minibuffer* specifies whether to include the minibuffer window in the returned list. If *minibuffer* is `t`, the minibuffer window is included. If *minibuffer* is `nil` or omitted, the minibuffer window is included only if it is active. If *minibuffer* is neither `nil` nor `t`, the minibuffer window is never included.

  The optional argument *window*, if non-`nil`, should be a live window on the specified frame; then *window* will be the first element in the returned list. If *window* is omitted or `nil`, the window selected within the frame is the first element.

Windows in the same frame are organized into a *window tree*, whose leaf nodes are the live windows. The internal nodes of a window tree are not live; they exist for the purpose of organizing the relationships between live windows. The root node of a window tree is called the *root window*. It can be either a live window (if the frame has just one window), or an internal window.

A minibuffer window (see ⟨undefined⟩ [Minibuffer Windows], page ⟨undefined⟩) is not part of its frame's window tree unless the frame is a minibuffer-only frame. Nonetheless, most of the functions in this section accept the minibuffer window as an argument. Also, the function `window-tree` described at the end of this section lists the minibuffer window alongside the actual window tree.

`frame-root-window` **&optional** *frame-or-window* [Function]
: This function returns the root window for *frame-or-window*. The argument *frame-or-window* should be either a window or a frame; if omitted or `nil`, it defaults to the selected frame. If *frame-or-window* is a window, the return value is the root window of that window's frame.

When a window is split, there are two live windows where previously there was one. One of these is represented by the same Lisp window object as the original window, and the other

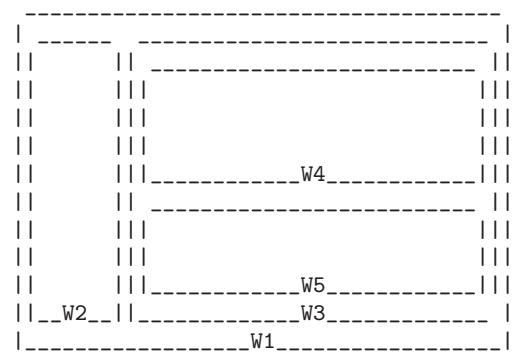
is represented by a newly-created Lisp window object. Both of these live windows become leaf nodes of the window tree, as *child windows* of a single internal window. If necessary, Emacs automatically creates this internal window, which is also called the *parent window*, and assigns it to the appropriate position in the window tree. A set of windows that share the same parent are called *siblings*.

`window-parent` **&optional** *window* [Function]
: This function returns the parent window of *window*. If *window* is omitted or `nil`, it defaults to the selected window. The return value is `nil` if *window* has no parent (i.e., it is a minibuffer window or the root window of its frame).

Each internal window always has at least two child windows. If this number falls to one as a result of window deletion, Emacs automatically deletes the internal window, and its sole remaining child window takes its place in the window tree.

Each child window can be either a live window, or an internal window (which in turn would have its own child windows). Therefore, each internal window can be thought of as occupying a certain rectangular *screen area*—the union of the areas occupied by the live windows that are ultimately descended from it.

For each internal window, the screen areas of the immediate children are arranged either vertically or horizontally (never both). If the child windows are arranged one above the other, they are said to form a *vertical combination*; if they are arranged side by side, they are said to form a *horizontal combination*. Consider the following example:




The root window of this frame is an internal window, *W1*. Its child windows form a horizontal combination, consisting of the live window *W2* and the internal window *W3*. The child windows of *W3* form a vertical combination, consisting of the live windows *W4* and *W5*. Hence, the live windows in this window tree are *W2* *W4*, and *W5*.

The following functions can be used to retrieve a child window of an internal window, and the siblings of a child window.

`window-top-child` *window* [Function]
: This function returns the topmost child window of *window*, if *window* is an internal window whose children form a vertical combination. For any other type of window, the return value is `nil`.

`window-left-child` *window* [Function]
: This function returns the leftmost child window of *window*, if *window* is an internal window whose children form a horizontal combination. For any other type of window, the return value is `nil`.

`window-child` *window* [Function]
: This function returns the first child window of the internal window *window*—the topmost child window for a vertical combination, or the leftmost child window for a horizontal combination. If *window* is a live window, the return value is `nil`.

`window-combined-p` **&optional** *window horizontal* [Function]
: This function returns a non-`nil` value if and only if *window* is part of a vertical combination. If *window* is omitted or `nil`, it defaults to the selected one.

  If the optional argument *horizontal* is non-`nil`, this means to return non-`nil` if and only if *window* is part of a horizontal combination.

`window-next-sibling` **&optional** *window* [Function]
: This function returns the next sibling of the window *window*. If omitted or `nil`, *window* defaults to the selected window. The return value is `nil` if *window* is the last child of its parent.

`window-prev-sibling` **&optional** *window* [Function]
: This function returns the previous sibling of the window *window*. If omitted or `nil`, *window* defaults to the selected window. The return value is `nil` if *window* is the first child of its parent.

The functions `window-next-sibling` and `window-prev-sibling` should not be confused with the functions `next-window` and `previous-window`, which return the next and previous window, respectively, in the cyclic ordering of windows (see Section 17.9 [Cyclic Window Ordering], page 307).

You can use the following functions to find the first live window on a frame and the window nearest to a given window.

`frame-first-window` **&optional** *frame-or-window* [Function]
: This function returns the live window at the upper left corner of the frame specified by *frame-or-window*. The argument *frame-or-window* must denote a window or a live frame and defaults to the selected frame. If *frame-or-window* specifies a window, this function returns the first window on that window's frame. Under the assumption that the frame from our canonical example is selected `(frame-first-window)` returns *W2*.

`window-in-direction` *direction* **&optional** *window ignore* [Function]
: This function returns the nearest live window in direction *direction* as seen from the position of `window-point` in window *window*. The argument *direction* must be one of `above`, `below`, `left` or `right`. The optional argument *window* must denote a live window and defaults to the selected one.

  This function does not return a window whose `no-other-window` parameter is non-`nil` (see Section 17.25 [Window Parameters], page 337). If the nearest window's `no-other-window` parameter is non-`nil`, this function tries to find another window

in the indicated direction whose `no-other-window` parameter is `nil`. If the optional argument *ignore* is non-`nil`, a window may be returned even if its `no-other-window` parameter is non-`nil`.

If it doesn't find a suitable window, this function returns `nil`.

The following function allows to retrieve the entire window tree of a frame:

`window-tree` **&optional** *frame* [Function]

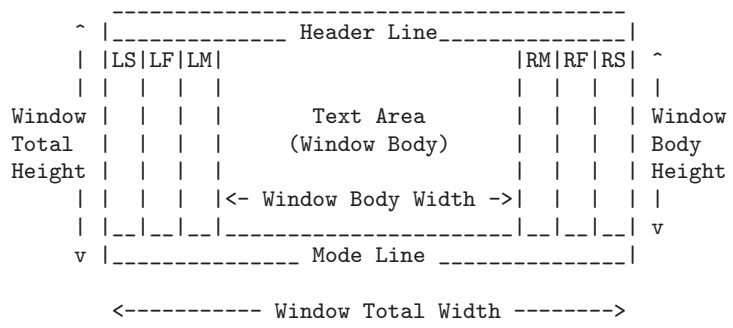
This function returns a list representing the window tree for frame *frame*. If *frame* is omitted or `nil`, it defaults to the selected frame.

The return value is a list of the form `(root mini)`, where *root* represents the window tree of the frame's root window, and *mini* is the frame's minibuffer window.

If the root window is live, *root* is that window itself. Otherwise, *root* is a list `(dir edges w1 w2 ...)` where *dir* is `nil` for a horizontal combination and `t` for a vertical combination, *edges* gives the size and position of the combination, and the remaining elements are the child windows. Each child window may again be a window object (for a live window) or a list with the same format as above (for an internal window). The *edges* element is a list `(left top right bottom)`, similar to the value returned by `window-edges` (see Section 17.23 [Coordinates and Windows], page 333).

## 17.3 Window Sizes

The following schematic shows the structure of a live window:




At the center of the window is the *text area*, or *body*, where the buffer text is displayed. On each side of the text area is a series of vertical areas; from innermost to outermost, these are the left and right margins, denoted by LM and RM in the schematic (see Section 11.15.5 [Display Margins], page 167); the left and right fringes, denoted by LF and RF (see Section 11.13 [Fringes], page 156); and the left or right scroll bar, only one of which is present at any time, denoted by LS and RS (see Section 11.14 [Scroll Bars], page 162). At the top of the window is an optional header line (see Section 20.4.7 [Header Lines], page 426), and at the bottom of the window is the mode line (see Section 20.4 [Mode Line Format], page 418).

Emacs provides several functions for finding the height and width of a window. Except where noted, Emacs reports window heights and widths as integer numbers of lines and columns, respectively. On a graphical display, each "line" and "column" actually corresponds to the height and width of a "default" character specified by the frame's default

font. Thus, if a window is displaying text with a different font or size, the reported height and width for that window may differ from the actual number of text lines or columns displayed within it.

The *total height* of a window is the distance between the top and bottom of the window, including the header line (if one exists) and the mode line. The *total width* of a window is the distance between the left and right edges of the mode line. Note that the height of a frame is not the same as the height of its windows, since a frame may also contain an echo area, menu bar, and tool bar (see Section 18.3.4 [Size and Position], page 354).

`window-total-height` **&optional** *window* [Function]
: This function returns the total height, in lines, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. If *window* is an internal window, the return value is the total height occupied by its descendant windows.

`window-total-width` **&optional** *window* [Function]
: This function returns the total width, in columns, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. If *window* is internal, the return value is the total width occupied by its descendant windows.

`window-total-size` **&optional** *window horizontal* [Function]
: This function returns either the total height or width of the window *window*. If *horizontal* is omitted or `nil`, this is equivalent to calling `window-total-height` for *window*; otherwise it is equivalent to calling `window-total-width` for *window*.

The following functions can be used to determine whether a given window has any adjacent windows.

`window-full-height-p` **&optional** *window* [Function]
: This function returns non-`nil` if *window* has no other window above or below it in its frame, i.e., its total height equals the total height of the root window on that frame. If *window* is omitted or `nil`, it defaults to the selected window.

`window-full-width-p` **&optional** *window* [Function]
: This function returns non-`nil` if *window* has no other window to the left or right in its frame, i.e., its total width equals that of the root window on that frame. If *window* is omitted or `nil`, it defaults to the selected window.

The *body height* of a window is the height of its text area, which does not include the mode or header line. Similarly, the *body width* is the width of the text area, which does not include the scroll bar, fringes, or margins.

`window-body-height` **&optional** *window* [Function]
: This function returns the body height, in lines, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window; otherwise it must be a live window.

  If there is a partially-visible line at the bottom of the text area, that counts as a whole line; to exclude such a partially-visible line, use `window-text-height`, below.

`window-body-width` **&optional** *window* [Function]
: This function returns the body width, in columns, of the window *window*. If *window* is omitted or `nil`, it defaults to the selected window; otherwise it must be a live window.

`window-body-size` **&optional** *window horizontal* [Function]
: This function returns the body height or body width of *window*. If *horizontal* is omitted or `nil`, it is equivalent to calling `window-body-height` for *window*; otherwise it is equivalent to calling `window-body-width`.

`window-text-height` **&optional** *window* [Function]
: This function is like `window-body-height`, except that any partially-visible line at the bottom of the text area is not counted.

For compatibility with previous versions of Emacs, `window-height` is an alias for `window-total-height`, and `window-width` is an alias for `window-body-width`. These aliases are considered obsolete and will be removed in the future.

Commands that change the size of windows (see Section 17.4 [Resizing Windows], page 295), or split them (see Section 17.5 [Splitting Windows], page 297), obey the variables `window-min-height` and `window-min-width`, which specify the smallest allowable window height and width. See Section "Deleting and Rearranging Windows" in *The GNU Emacs Manual*. They also obey the variable `window-size-fixed`, with which a window can be *fixed* in size:

`window-size-fixed` [Variable]
: If this buffer-local variable is non-`nil`, the size of any window displaying the buffer cannot normally be changed. Deleting a window or changing the frame's size may still change its size, if there is no choice.

  If the value is `height`, then only the window's height is fixed; if the value is `width`, then only the window's width is fixed. Any other non-`nil` value fixes both the width and the height.

`window-size-fixed-p` **&optional** *window horizontal* [Function]
: This function returns a non-`nil` value if *window*'s height is fixed. If *window* is omitted or `nil`, it defaults to the selected window. If the optional argument *horizontal* is non-`nil`, the return value is non-`nil` if *window*'s width is fixed.

  A `nil` return value does not necessarily mean that *window* can be resized in the desired direction. To determine that, use the function `window-resizable`. See Section 17.4 [Resizing Windows], page 295.

See Section 17.23 [Coordinates and Windows], page 333, for more functions that report the positions of various parts of a window relative to the frame, from which you can calculate its size. In particular, you can use the functions `window-pixel-edges` and `window-inside-pixel-edges` to find the size in pixels, for graphical displays.

## 17.4 Resizing Windows

This section describes functions for resizing a window without changing the size of its frame. Because live windows do not overlap, these functions are meaningful only on frames that contain two or more windows: resizing a window also changes the size of a neighboring window. If there is just one window on a frame, its size cannot be changed except by resizing the frame (see Section 18.3.4 [Size and Position], page 354).

Except where noted, these functions also accept internal windows as arguments. Resizing an internal window causes its child windows to be resized to fit the same space.

`window-resizable` *window delta* **&optional** *horizontal ignore* [Function]
This function returns *delta* if the size of *window* can be changed vertically by *delta* lines. If the optional argument *horizontal* is non-`nil`, it instead returns *delta* if *window* can be resized horizontally by *delta* columns. It does not actually change the window size.

If *window* is `nil`, it defaults to the selected window.

A positive value of *delta* means to check whether the window can be enlarged by that number of lines or columns; a negative value of *delta* means to check whether the window can be shrunk by that many lines or columns. If *delta* is non-zero, a return value of 0 means that the window cannot be resized.

Normally, the variables `window-min-height` and `window-min-width` specify the smallest allowable window size. See Section "Deleting and Rearranging Windows" in *The GNU Emacs Manual*. However, if the optional argument *ignore* is non-`nil`, this function ignores `window-min-height` and `window-min-width`, as well as `window-size-fixed`. Instead, it considers the minimum-height window to be one consisting of a header (if any), a mode line, plus a text area one line tall; and a minimum-width window as one consisting of fringes, margins, and scroll bar (if any), plus a text area two columns wide.

`window-resize` *window delta* **&optional** *horizontal ignore* [Function]
This function resizes *window* by *delta* increments. If *horizontal* is `nil`, it changes the height by *delta* lines; otherwise, it changes the width by *delta* columns. A positive *delta* means to enlarge the window, and a negative *delta* means to shrink it.

If *window* is `nil`, it defaults to the selected window. If the window cannot be resized as demanded, an error is signaled.

The optional argument *ignore* has the same meaning as for the function `window-resizable` above.

The choice of which window edges this function alters depends on the values of the option `window-combination-resize` and the combination limits of the involved windows; in some cases, it may alter both edges. See Section 17.7 [Recombining Windows], page 301. To resize by moving only the bottom or right edge of a window, use the function `adjust-window-trailing-edge`, below.

`adjust-window-trailing-edge` *window delta* **&optional** *horizontal* [Function]
This function moves *window*'s bottom edge by *delta* lines. If optional argument *horizontal* is non-`nil`, it instead moves the right edge by *delta* columns. If *window* is `nil`, it defaults to the selected window.

A positive *delta* moves the edge downwards or to the right; a negative *delta* moves it upwards or to the left. If the edge cannot be moved as far as specified by *delta*, this function moves it as far as possible but does not signal a error.

This function tries to resize windows adjacent to the edge that is moved. If this is not possible for some reason (e.g., if that adjacent window is fixed-size), it may resize other windows.

The following commands resize windows in more specific ways. When called interactively, they act on the selected window.

`fit-window-to-buffer` **&optional** *window max-height min-height override* [Command]

This command adjusts the height of *window* to fit the text in it. It returns non-`nil` if it was able to resize *window*, and `nil` otherwise. If *window* is omitted or `nil`, it defaults to the selected window. Otherwise, it should be a live window.

The optional argument *max-height*, if non-`nil`, specifies the maximum total height that this function can give *window*. The optional argument *min-height*, if non-`nil`, specifies the minimum total height that it can give, which overrides the variable `window-min-height`.

If the optional argument *override* is non-`nil`, this function ignores any size restrictions imposed by `window-min-height` and `window-min-width`.

If the option `fit-frame-to-buffer` is non-`nil`, this command may resize the frame to fit its contents.

`shrink-window-if-larger-than-buffer` **&optional** *window* [Command]

This command attempts to reduce *window*'s height as much as possible while still showing its full buffer, but no less than `window-min-height` lines. The return value is non-`nil` if the window was resized, and `nil` otherwise. If *window* is omitted or `nil`, it defaults to the selected window. Otherwise, it should be a live window.

This command does nothing if the window is already too short to display all of its buffer, or if any of the buffer is scrolled off-screen, or if the window is the only live window in its frame.

`balance-windows` **&optional** *window-or-frame* [Command]

This function balances windows in a way that gives more space to full-width and/or full-height windows. If *window-or-frame* specifies a frame, it balances all windows on that frame. If *window-or-frame* specifies a window, it balances only that window and its siblings (see Section 17.2 [Windows and Frames], page 290).

`balance-windows-area` [Command]

This function attempts to give all windows on the selected frame approximately the same share of the screen area. Full-width or full-height windows are not given more space than other windows.

`maximize-window` **&optional** *window* [Command]

This function attempts to make *window* as large as possible, in both dimensions, without resizing its frame or deleting other windows. If *window* is omitted or `nil`, it defaults to the selected window.

`minimize-window` **&optional** *window* [Command]

This function attempts to make *window* as small as possible, in both dimensions, without deleting it or resizing its frame. If *window* is omitted or `nil`, it defaults to the selected window.

## 17.5 Splitting Windows

This section describes functions for creating a new window by *splitting* an existing one.

`split-window` **&optional** *window size side* [Command]
This function creates a new live window next to the window *window*. If *window* is omitted or `nil`, it defaults to the selected window. That window is "split", and reduced in size. The space is taken up by the new window, which is returned.

The optional second argument *size* determines the sizes of *window* and/or the new window. If it is omitted or `nil`, both windows are given equal sizes; if there is an odd line, it is allocated to the new window. If *size* is a positive number, *window* is given *size* lines (or columns, depending on the value of *side*). If *size* is a negative number, the new window is given $-$*size* lines (or columns).

If *size* is `nil`, this function obeys the variables `window-min-height` and `window-min-width`. See Section "Deleting and Rearranging Windows" in *The GNU Emacs Manual*. Thus, it signals an error if splitting would result in making a window smaller than those variables specify. However, a non-`nil` value for *size* causes those variables to be ignored; in that case, the smallest allowable window is considered to be one that has space for a text area one line tall and/or two columns wide.

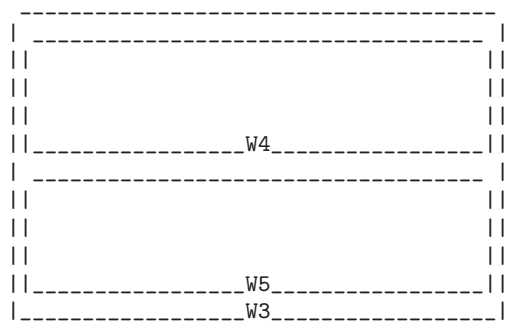
The optional third argument *side* determines the position of the new window relative to *window*. If it is `nil` or `below`, the new window is placed below *window*. If it is `above`, the new window is placed above *window*. In both these cases, *size* specifies a total window height, in lines.

If *side* is `t` or `right`, the new window is placed on the right of *window*. If *side* is `left`, the new window is placed on the left of *window*. In both these cases, *size* specifies a total window width, in columns.

If *window* is a live window, the new window inherits various properties from it, including margins and scroll bars. If *window* is an internal window, the new window inherits the properties of the window selected within *window*'s frame.

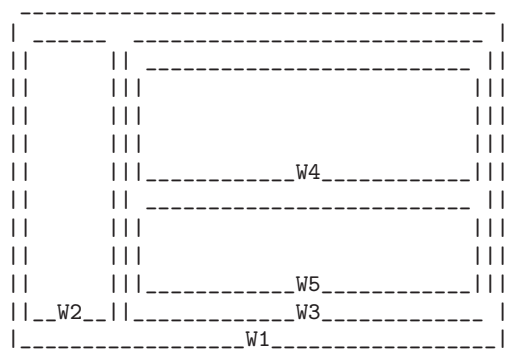
The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `split-window` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `split-window` window parameter is a function, that function is called with the arguments *window*, *size*, and *side*, in lieu of the usual action of `split-window`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 17.25 [Window Parameters], page 337.

As an example, here is a sequence of `split-window` calls that yields the window configuration discussed in Section 17.2 [Windows and Frames], page 290. This example demonstrates splitting a live window as well as splitting an internal window. We begin with a frame containing a single window (a live root window), which we denote by *W4*. Calling `(split-window W4)` yields this window configuration:




The `split-window` call has created a new live window, denoted by *W5*. It has also created a new internal window, denoted by *W3*, which becomes the root window and the parent of both *W4* and *W5*.

Next, we call `(split-window W3 nil 'left)`, passing the internal window *W3* as the argument. The result:




A new live window *W2* is created, to the left of the internal window *W3*. A new internal window *W1* is created, becoming the new root window.

For interactive use, Emacs provides two commands which always split the selected window. These call `split-window` internally.

`split-window-right` **&optional** *size* [Command]
: This function splits the selected window into two side-by-side windows, putting the selected window on the left. If *size* is positive, the left window gets *size* columns; if *size* is negative, the right window gets −*size* columns.

`split-window-below` **&optional** *size* [Command]
: This function splits the selected window into two windows, one above the other, leaving the upper window selected. If *size* is positive, the upper window gets *size* lines; if *size* is negative, the lower window gets −*size* lines.

`split-window-keep-point` [User Option]
: If the value of this variable is non-`nil` (the default), `split-window-below` behaves as described above.

  If it is `nil`, `split-window-below` adjusts point in each of the two windows to minimize redisplay. (This is useful on slow terminals.) It selects whichever window contains the

screen line that point was previously on. Note that this only affects `split-window-below`, not the lower-level `split-window` function.

## 17.6 Deleting Windows

*Deleting* a window removes it from the frame's window tree. If the window is a live window, it disappears from the screen. If the window is an internal window, its child windows are deleted too.

Even after a window is deleted, it continues to exist as a Lisp object, until there are no more references to it. Window deletion can be reversed, by restoring a saved window configuration (see Section 17.24 [Window Configurations], page 335).

`delete-window` **&optional** *window* [Command]

This function removes *window* from display and returns `nil`. If *window* is omitted or `nil`, it defaults to the selected window. If deleting the window would leave no more windows in the window tree (e.g., if it is the only live window in the frame), an error is signaled.

By default, the space taken up by *window* is given to one of its adjacent sibling windows, if any. However, if the variable `window-combination-resize` is non-`nil`, the space is proportionally distributed among any remaining windows in the window combination. See Section 17.7 [Recombining Windows], page 301.

The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `delete-window` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `delete-window` window parameter is a function, that function is called with the argument *window*, in lieu of the usual action of `delete-window`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 17.25 [Window Parameters], page 337.

`delete-other-windows` **&optional** *window* [Command]

This function makes *window* fill its frame, by deleting other windows as necessary. If *window* is omitted or `nil`, it defaults to the selected window. The return value is `nil`.

The behavior of this function may be altered by the window parameters of *window*, so long as the variable `ignore-window-parameters` is `nil`. If the value of the `delete-other-windows` window parameter is `t`, this function ignores all other window parameters. Otherwise, if the value of the `delete-other-windows` window parameter is a function, that function is called with the argument *window*, in lieu of the usual action of `delete-other-windows`. Otherwise, this function obeys the `window-atom` or `window-side` window parameter, if any. See Section 17.25 [Window Parameters], page 337.

`delete-windows-on` **&optional** *buffer-or-name frame* [Command]

This function deletes all windows showing *buffer-or-name*, by calling `delete-window` on those windows. *buffer-or-name* should be a buffer, or the name of a buffer; if omitted or `nil`, it defaults to the current buffer. If there are no windows showing the specified buffer, this function does nothing. If the specified buffer is a minibuffer, an error is signaled.

If there is a dedicated window showing the buffer, and that window is the only one on its frame, this function also deletes that frame if it is not the only frame on the terminal.

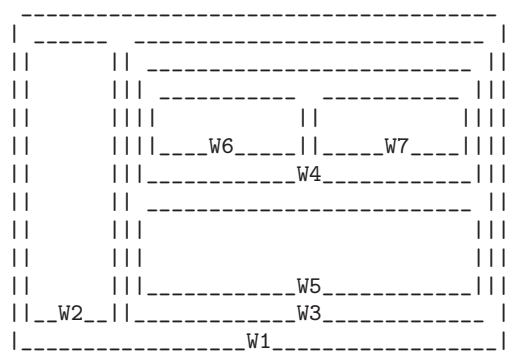
The optional argument *frame* specifies which frames to operate on:

- `nil` means operate on all frames.
- `t` means operate on the selected frame.
- `visible` means operate on all visible frames.
- `0` means operate on all visible or iconified frames.
- A frame means operate on that frame.

Note that this argument does not have the same meaning as in other functions which scan all live windows (see Section 17.9 [Cyclic Window Ordering], page 307). Specifically, the meanings of `t` and `nil` here are the opposite of what they are in those other functions.

## 17.7 Recombining Windows

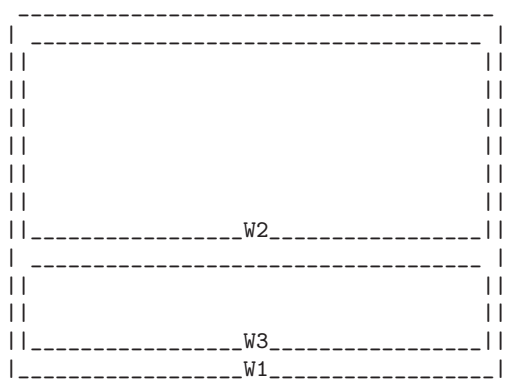
When deleting the last sibling of a window *W*, its parent window is deleted too, with *W* replacing it in the window tree. This means that *W* must be recombined with its parent's siblings to form a new window combination (see Section 17.2 [Windows and Frames], page 290). In some occasions, deleting a live window may even entail the deletion of two internal windows.



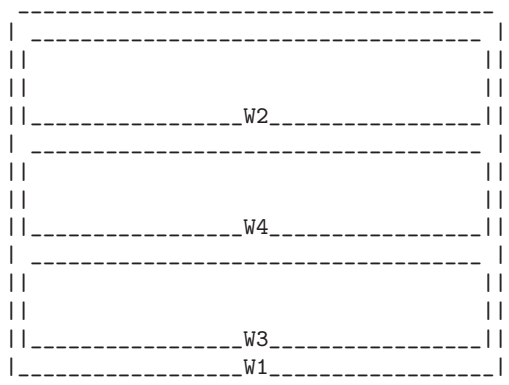

Deleting *W5* in this configuration normally causes the deletion of *W3* and *W4*. The remaining live windows *W2*, *W6* and *W7* are recombined to form a new horizontal combination with parent *W1*.

Sometimes, however, it makes sense to not delete a parent window like *W4*. In particular, a parent window should not be removed when it was used to preserve a combination embedded in a combination of the same type. Such embeddings make sense to assure that when you split a window and subsequently delete the new window, Emacs reestablishes the layout of the associated frame as it existed before the splitting.

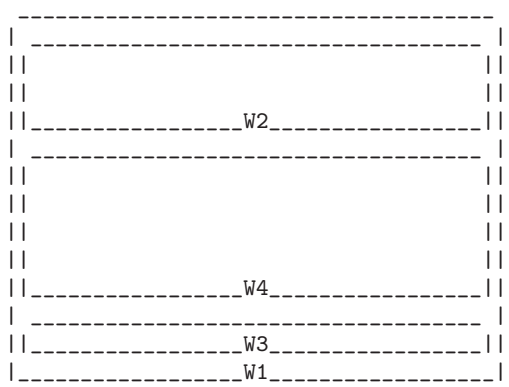
Consider a scenario starting with two live windows *W2* and *W3* and their parent *W1*.



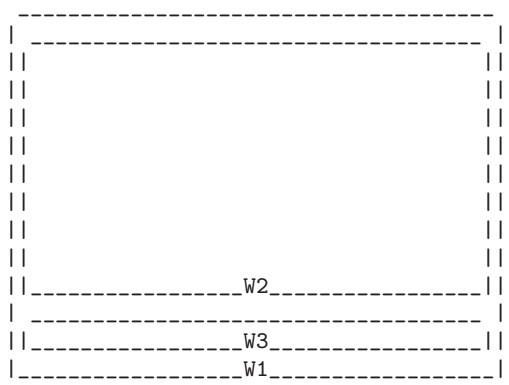

Split *W2* to make a new window *W4* as follows.




Now, when enlarging a window vertically, Emacs tries to obtain the corresponding space from its lower sibling, provided such a window exists. In our scenario, enlarging *W4* will steal space from *W3*.




Deleting *W4* will now give its entire space to *W2*, including the space earlier stolen from *W3*.




This can be counterintutive, in particular if *W4* were used for displaying a buffer only temporarily (see Section 11.8 [Temporary Displays], page 125), and you want to continue working with the initial layout.

The behavior can be fixed by making a new parent window when splitting *W2*. The variable described next allows to do that.

`window-combination-limit` [User Option]

This variable controls whether splitting a window shall make a new parent window. The following values are recognized:

`nil` This means that the new live window is allowed to share the existing parent window, if one exists, provided the split occurs in the same direction as the existing window combination (otherwise, a new internal window is created anyway).

`window-size`
In this case `display-buffer` makes a new parent window if it is passed a `window-height` or `window-width` entry in the *alist* argument (see Section 17.13 [Display Action Functions], page 314).

`temp-buffer`
This value causes the creation of a new parent window when a window is split for showing a temporary buffer (see Section 11.8 [Temporary Displays], page 125) only.

`display-buffer`
This means that when `display-buffer` (see Section 17.12 [Choosing Window], page 313) splits a window it always makes a new parent window.

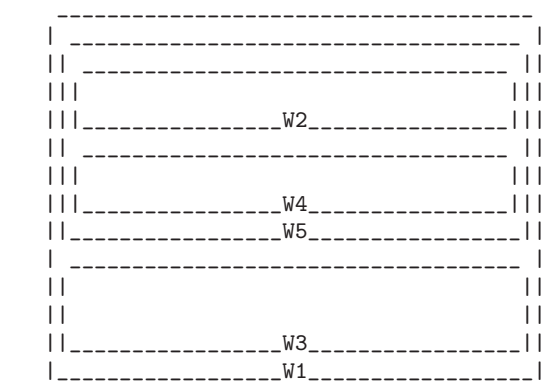
`t` In this case a new parent window is always created when splitting a window. Thus, if the value of this variable is at all times `t`, then at all times every window tree is a binary tree (a tree where each window except the root window has exactly one sibling).

The default is `nil`. Other values are reserved for future use.

If, as a consequence of this variable's setting, `split-window` makes a new parent window, it also calls `set-window-combination-limit` (see below) on the newly-created

internal window. This affects how the window tree is rearranged when the child windows are deleted (see below).

If `window-combination-limit` is `t`, splitting *W2* in the initial configuration of our scenario would have produced this:




A new internal window *W5* has been created; its children are *W2* and the new live window *W4*. Now, *W2* is the only sibling of *W4*, so enlarging *W4* will try to shrink *W2*, leaving *W3* unaffected. Observe that *W5* represents a vertical combination of two windows embedded in the vertical combination *W1*.

`set-window-combination-limit` *window limit* [Function]
: This functions sets the *combination limit* of the window *window* to *limit*. This value can be retrieved via the function `window-combination-limit`. See below for its effects; note that it is only meaningful for internal windows. The `split-window` function automatically calls this function, passing it `t` as *limit*, provided the value of the variable `window-combination-limit` is `t` when it is called.

`window-combination-limit` *window* [Function]
: This function returns the combination limit for *window*.

  The combination limit is meaningful only for an internal window. If it is `nil`, then Emacs is allowed to automatically delete *window*, in response to a window deletion, in order to group the child windows of *window* with its sibling windows to form a new window combination. If the combination limit is `t`, the child windows of *window* are never automatically recombined with its siblings.

  If, in the configuration shown at the beginning of this section, the combination limit of *W4* (the parent window of *W6* and *W7*) is `t`, deleting *W5* will not implicitly delete *W4* too.

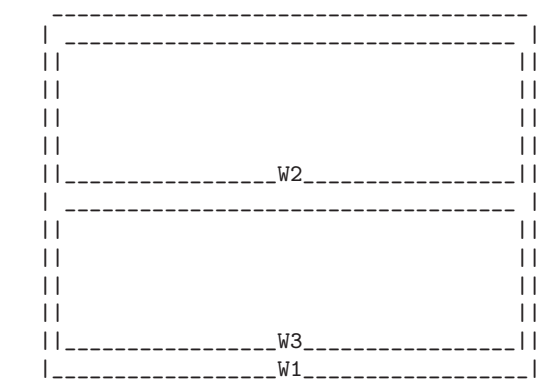
Alternatively, the problems sketched above can be avoided by always resizing all windows in the same combination whenever one of its windows is split or deleted. This also permits to split windows that would be otherwise too small for such an operation.

`window-combination-resize` [User Option]
: If this variable is `nil`, `split-window` can only split a window (denoted by *window*) if *window*'s screen area is large enough to accommodate both itself and the new window.

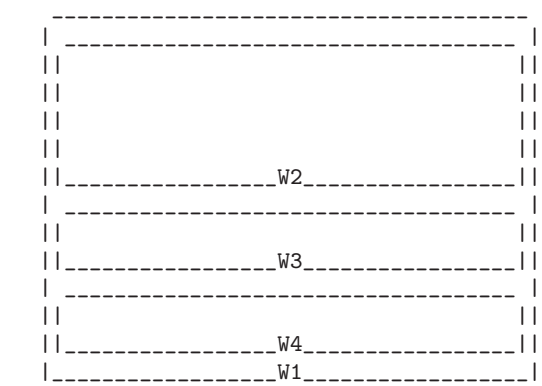
If this variable is `t`, `split-window` tries to resize all windows that are part of the same combination as *window*, in order to accommodate the new window. In particular, this may allow `split-window` to succeed even if *window* is a fixed-size window or too small to ordinarily split. Furthermore, subsequently resizing or deleting *window* may resize all other windows in its combination.

The default is `nil`. Other values are reserved for future use. The value of this variable is ignored when `window-combination-limit` is non-`nil`.

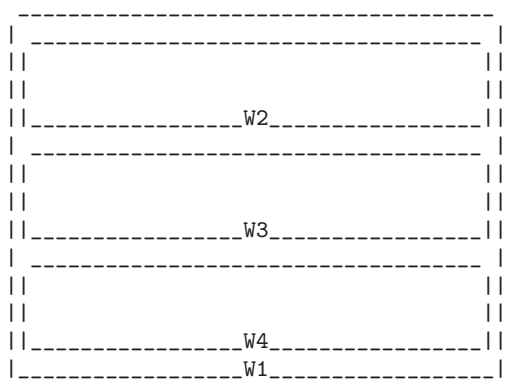
To illustrate the effect of `window-combination-resize`, consider the following frame layout.




If `window-combination-resize` is `nil`, splitting window *W3* leaves the size of *W2* unchanged:




If `window-combination-resize` is `t`, splitting *W3* instead leaves all three live windows with approximately the same height:




Deleting any of the live windows *W2*, *W3* or *W4* will distribute its space proportionally among the two remaining live windows.

## 17.8 Selecting Windows

`select-window` *window* **&optional** *norecord* [Function]

This function makes *window* the selected window and the window selected within its frame (see Section 17.1 [Basic Windows], page 289) and selects that frame. *window* must be a live window. This function also makes *window*'s buffer (see Section 17.10 [Buffers and Windows], page 309) current and sets that buffer's value of `point` to the value of `window-point` (see Section 17.18 [Window Point], page 323) in *window*. The return value is *window*.

By default, this function also moves *window*'s buffer to the front of the buffer list (see Section 16.8 [The Buffer List], page 281), and makes *window* the most recently selected window. However, if the optional argument *norecord* is non-`nil`, these additional actions are omitted.

The sequence of calls to `select-window` with a non-`nil` *norecord* argument determines an ordering of windows by their selection time. The function `get-lru-window` can be used to retrieve the least recently selected live window (see Section 17.9 [Cyclic Window Ordering], page 307).

`save-selected-window` *forms...* [Macro]

This macro records the selected frame, as well as the selected window of each frame, executes *forms* in sequence, then restores the earlier selected frame and windows. It also saves and restores the current buffer. It returns the value of the last form in *forms*.

This macro does not save or restore anything about the sizes, arrangement or contents of windows; therefore, if *forms* change them, the change persists. If the previously selected window of some frame is no longer live at the time of exit from *forms*, that frame's selected window is left alone. If the previously selected window is no longer live, then whatever window is selected at the end of *forms* remains selected. The current buffer is restored if and only if it is still live when exiting *forms*.

This macro changes neither the ordering of recently selected windows nor the buffer list.

`with-selected-window` *window forms...* [Macro]
This macro selects *window*, executes *forms* in sequence, then restores the previously selected window and current buffer. The ordering of recently selected windows and the buffer list remain unchanged unless you deliberately change them within *forms*; for example, by calling `select-window` with argument *norecord* `nil`.

This macro does not change the order of recently selected windows or the buffer list.

`frame-selected-window` **&optional** *frame* [Function]
This function returns the window on *frame* that is selected within that frame. *frame* should be a live frame; if omitted or `nil`, it defaults to the selected frame.

`set-frame-selected-window` *frame window* **&optional** *norecord* [Function]
This function makes *window* the window selected within the frame *frame*. *frame* should be a live frame; if omitted or `nil`, it defaults to the selected frame. *window* should be a live window; if omitted or `nil`, it defaults to the selected window.

If *frame* is the selected frame, this makes *window* the selected window.

If the optional argument *norecord* is non-`nil`, this function does not alter the list of most recently selected windows, nor the buffer list.

## 17.9 Cyclic Ordering of Windows

When you use the command `C-x o` (`other-window`) to select some other window, it moves through live windows in a specific order. For any given configuration of windows, this order never varies. It is called the *cyclic ordering of windows*.

The ordering is determined by a depth-first traversal of the frame's window tree, retrieving the live windows which are the leaf nodes of the tree (see Section 17.2 [Windows and Frames], page 290). If the minibuffer is active, the minibuffer window is included too. The ordering is cyclic, so the last window in the sequence is followed by the first one.

`next-window` **&optional** *window minibuf all-frames* [Function]
This function returns a live window, the one following *window* in the cyclic ordering of windows. *window* should be a live window; if omitted or `nil`, it defaults to the selected window.

The optional argument *minibuf* specifies whether minibuffer windows should be included in the cyclic ordering. Normally, when *minibuf* is `nil`, a minibuffer window is included only if it is currently "active"; this matches the behavior of `C-x o`. (Note that a minibuffer window is active as long as its minibuffer is in use; see ⟨undefined⟩ [Minibuffers], page ⟨undefined⟩).

If *minibuf* is `t`, the cyclic ordering includes all minibuffer windows. If *minibuf* is neither `t` nor `nil`, minibuffer windows are not included even if they are active.

The optional argument *all-frames* specifies which frames to consider:

- `nil` means to consider windows on *window*'s frame. If the minibuffer window is considered (as specified by the *minibuf* argument), then frames that share the minibuffer window are considered too.
- `t` means to consider windows on all existing frames.
- `visible` means to consider windows on all visible frames.

- 0 means to consider windows on all visible or iconified frames.
- A frame means to consider windows on that specific frame.
- Anything else means to consider windows on *window*'s frame, and no others.

If more than one frame is considered, the cyclic ordering is obtained by appending the orderings for those frames, in the same order as the list of all live frames (see Section 18.7 [Finding All Frames], page 357).

`previous-window` **&optional** *window minibuf all-frames* [Function]
This function returns a live window, the one preceding *window* in the cyclic ordering of windows. The other arguments are handled like in `next-window`.

`other-window` *count* **&optional** *all-frames* [Command]
This function selects a live window, one *count* places from the selected window in the cyclic ordering of windows. If *count* is a positive number, it skips *count* windows forwards; if *count* is negative, it skips −*count* windows backwards; if *count* is zero, that simply re-selects the selected window. When called interactively, *count* is the numeric prefix argument.

The optional argument *all-frames* has the same meaning as in `next-window`, like a `nil` *minibuf* argument to `next-window`.

This function does not select a window that has a non-`nil` `no-other-window` window parameter (see Section 17.25 [Window Parameters], page 337).

`walk-windows` *fun* **&optional** *minibuf all-frames* [Function]
This function calls the function *fun* once for each live window, with the window as the argument.

It follows the cyclic ordering of windows. The optional arguments *minibuf* and *all-frames* specify the set of windows included; these have the same arguments as in `next-window`. If *all-frames* specifies a frame, the first window walked is the first window on that frame (the one returned by `frame-first-window`), not necessarily the selected window.

If *fun* changes the window configuration by splitting or deleting windows, that does not alter the set of windows walked, which is determined prior to calling *fun* for the first time.

`one-window-p` **&optional** *no-mini all-frames* [Function]
This function returns `t` if the selected window is the only live window, and `nil` otherwise.

If the minibuffer window is active, it is normally considered (so that this function returns `nil`). However, if the optional argument *no-mini* is non-`nil`, the minibuffer window is ignored even if active. The optional argument *all-frames* has the same meaning as for `next-window`.

The following functions return a window which satisfies some criterion, without selecting it:

`get-lru-window` **&optional** *all-frames dedicated not-selected* [Function]
This function returns a live window which is heuristically the "least recently used" window. The optional argument *all-frames* has the same meaning as in `next-window`.

If any full-width windows are present, only those windows are considered. A minibuffer window is never a candidate. A dedicated window (see Section 17.16 [Dedicated Windows], page 321) is never a candidate unless the optional argument *dedicated* is non-`nil`. The selected window is never returned, unless it is the only candidate. However, if the optional argument *not-selected* is non-`nil`, this function returns `nil` in that case.

`get-largest-window` **&optional** *all-frames dedicated not-selected* [Function]
This function returns the window with the largest area (height times width). The optional argument *all-frames* specifies the windows to search, and has the same meaning as in `next-window`.

A minibuffer window is never a candidate. A dedicated window (see Section 17.16 [Dedicated Windows], page 321) is never a candidate unless the optional argument *dedicated* is non-`nil`. The selected window is not a candidate if the optional argument *not-selected* is non-`nil`. If the optional argument *not-selected* is non-`nil` and the selected window is the only candidate, this function returns `nil`.

If there are two candidate windows of the same size, this function prefers the one that comes first in the cyclic ordering of windows, starting from the selected window.

`get-window-with-predicate` *predicate* **&optional** *minibuf all-frames default* [Function]
This function calls the function *predicate* for each of the windows in the cyclic order of windows in turn, passing it the window as an argument. If the predicate returns non-`nil` for any window, this function stops and returns that window. If no such window is found, the return value is *default* (which defaults to `nil`).

The optional arguments *minibuf* and *all-frames* specify the windows to search, and have the same meanings as in `next-window`.

## 17.10 Buffers and Windows

This section describes low-level functions for examining and setting the contents of windows. See Section 17.11 [Switching Buffers], page 311, for higher-level functions for displaying a specific buffer in a window.

`window-buffer` **&optional** *window* [Function]
This function returns the buffer that *window* is displaying. If *window* is omitted or `nil` it defaults to the selected window. If *window* is an internal window, this function returns `nil`.

`set-window-buffer` *window buffer-or-name* **&optional** *keep-margins* [Function]
This function makes *window* display *buffer-or-name*. *window* should be a live window; if `nil`, it defaults to the selected window. *buffer-or-name* should be a buffer, or the name of an existing buffer. This function does not change which window is selected, nor does it directly change which buffer is current (see Section 16.2 [Current Buffer], page 272). Its return value is `nil`.

If *window* is *strongly dedicated* to a buffer and *buffer-or-name* does not specify that buffer, this function signals an error. See Section 17.16 [Dedicated Windows], page 321.

By default, this function resets *window*'s position, display margins, fringe widths, and scroll bar settings, based on the local variables in the specified buffer. However, if the optional argument *keep-margins* is non-`nil`, it leaves the display margins and fringe widths unchanged.

When writing an application, you should normally use the higher-level functions described in Section 17.11 [Switching Buffers], page 311, instead of calling `set-window-buffer` directly.

This runs `window-scroll-functions`, followed by `window-configuration-change-hook`. See Section 17.26 [Window Hooks], page 339.

`buffer-display-count` [Variable]

This buffer-local variable records the number of times a buffer has been displayed in a window. It is incremented each time `set-window-buffer` is called for the buffer.

`buffer-display-time` [Variable]

This buffer-local variable records the time at which a buffer was last displayed in a window. The value is `nil` if the buffer has never been displayed. It is updated each time `set-window-buffer` is called for the buffer, with the value returned by `current-time` (see ⟨undefined⟩ [Time of Day], page ⟨undefined⟩).

`get-buffer-window` **&optional** *buffer-or-name all-frames* [Function]

This function returns the first window displaying *buffer-or-name* in the cyclic ordering of windows, starting from the selected window (see Section 17.9 [Cyclic Window Ordering], page 307). If no such window exists, the return value is `nil`.

*buffer-or-name* should be a buffer or the name of a buffer; if omitted or `nil`, it defaults to the current buffer. The optional argument *all-frames* specifies which windows to consider:

- `t` means consider windows on all existing frames.
- `visible` means consider windows on all visible frames.
- 0 means consider windows on all visible or iconified frames.
- A frame means consider windows on that frame only.
- Any other value means consider windows on the selected frame.

Note that these meanings differ slightly from those of the *all-frames* argument to `next-window` (see Section 17.9 [Cyclic Window Ordering], page 307). This function may be changed in a future version of Emacs to eliminate this discrepancy.

`get-buffer-window-list` **&optional** *buffer-or-name minibuf all-frames* [Function]

This function returns a list of all windows currently displaying *buffer-or-name*. *buffer-or-name* should be a buffer or the name of an existing buffer. If omitted or `nil`, it defaults to the current buffer.

The arguments *minibuf* and *all-frames* have the same meanings as in the function `next-window` (see Section 17.9 [Cyclic Window Ordering], page 307). Note that the *all-frames* argument does *not* behave exactly like in `get-buffer-window`.

`replace-buffer-in-windows` **&optional** *buffer-or-name* [Command]

This command replaces *buffer-or-name* with some other buffer, in all windows displaying it. *buffer-or-name* should be a buffer, or the name of an existing buffer; if omitted or `nil`, it defaults to the current buffer.

The replacement buffer in each window is chosen via `switch-to-prev-buffer` (see Section 17.15 [Window History], page 319). Any dedicated window displaying *buffer-or-name* is deleted if possible (see Section 17.16 [Dedicated Windows], page 321). If such a window is the only window on its frame and there are other frames on the same terminal, the frame is deleted as well. If the dedicated window is the only window on the only frame on its terminal, the buffer is replaced anyway.

## 17.11 Switching to a Buffer in a Window

This section describes high-level functions for switching to a specified buffer in some window. In general, "switching to a buffer" means to (1) show the buffer in some window, (2) make that window the selected window (and its frame the selected frame), and (3) make the buffer the current buffer.

Do *not* use these functions to make a buffer temporarily current just so a Lisp program can access or modify it. They have side-effects, such as changing window histories (see Section 17.15 [Window History], page 319), which will surprise the user if used that way. If you want to make a buffer current to modify it in Lisp, use `with-current-buffer`, `save-current-buffer`, or `set-buffer`. See Section 16.2 [Current Buffer], page 272.

`switch-to-buffer` *buffer-or-name* **&optional** *norecord* [Command]
*force-same-window*

This command attempts to display *buffer-or-name* in the selected window and make it the current buffer. It is often used interactively (as the binding of `C-x b`), as well as in Lisp programs. The return value is the buffer switched to.

If *buffer-or-name* is `nil`, it defaults to the buffer returned by `other-buffer` (see Section 16.8 [The Buffer List], page 281). If *buffer-or-name* is a string that is not the name of any existing buffer, this function creates a new buffer with that name; the new buffer's major mode is determined by the variable `major-mode` (see Section 20.2 [Major Modes], page 399).

Normally, the specified buffer is put at the front of the buffer list—both the global buffer list and the selected frame's buffer list (see Section 16.8 [The Buffer List], page 281). However, this is not done if the optional argument *norecord* is non-`nil`.

Sometimes, `switch-to-buffer` may be unable to display the buffer in the selected window. This happens if the selected window is a minibuffer window, or if the selected window is strongly dedicated to its buffer (see Section 17.16 [Dedicated Windows], page 321). In that case, the command normally tries to display the buffer in some other window, by invoking `pop-to-buffer` (see below). However, if the optional argument *force-same-window* is non-`nil`, it signals an error instead.

By default, `switch-to-buffer` shows the buffer at its position of `point`. This behavior can be tuned using the following option.

`switch-to-buffer-preserve-window-point` [User Option]
: If this variable is `nil`, `switch-to-buffer` displays the buffer specified by *buffer-or-name* at the position of that buffer's `point`. If this variable is `already-displayed`, it tries to display the buffer at its previous position in the selected window, provided the buffer is currently displayed in some other window on any visible or iconified frame. If this variable is `t`, `switch-to-buffer` unconditionally tries to display the buffer at its previous position in the selected window.

  This variable is ignored if the buffer is already displayed in the selected window or never appeared in it before, or if `switch-to-buffer` calls `pop-to-buffer` to display the buffer.

The next two commands are similar to `switch-to-buffer`, except for the described features.

`switch-to-buffer-other-window` *buffer-or-name* **&optional** *norecord* [Command]
: This function displays the buffer specified by *buffer-or-name* in some window other than the selected window. It uses the function `pop-to-buffer` internally (see below).

  If the selected window already displays the specified buffer, it continues to do so, but another window is nonetheless found to display it as well.

  The *buffer-or-name* and *norecord* arguments have the same meanings as in `switch-to-buffer`.

`switch-to-buffer-other-frame` *buffer-or-name* **&optional** *norecord* [Command]
: This function displays the buffer specified by *buffer-or-name* in a new frame. It uses the function `pop-to-buffer` internally (see below).

  If the specified buffer is already displayed in another window, in any frame on the current terminal, this switches to that window instead of creating a new frame. However, the selected window is never used for this.

  The *buffer-or-name* and *norecord* arguments have the same meanings as in `switch-to-buffer`.

The above commands use the function `pop-to-buffer`, which flexibly displays a buffer in some window and selects that window for editing. In turn, `pop-to-buffer` uses `display-buffer` for displaying the buffer. Hence, all the variables affecting `display-buffer` will affect it as well. See Section 17.12 [Choosing Window], page 313, for the documentation of `display-buffer`.

`pop-to-buffer` *buffer-or-name* **&optional** *action norecord* [Command]
: This function makes *buffer-or-name* the current buffer and displays it in some window, preferably not the window previously selected. It then selects the displaying window. If that window is on a different graphical frame, that frame is given input focus if possible (see Section 18.9 [Input Focus], page 358). The return value is the buffer that was switched to.

  If *buffer-or-name* is `nil`, it defaults to the buffer returned by `other-buffer` (see Section 16.8 [The Buffer List], page 281). If *buffer-or-name* is a string that is not the name of any existing buffer, this function creates a new buffer with that name; the new buffer's major mode is determined by the variable `major-mode` (see Section 20.2 [Major Modes], page 399).

If *action* is non-`nil`, it should be a display action to pass to `display-buffer` (see Section 17.12 [Choosing Window], page 313). Alternatively, a non-`nil`, non-list value means to pop to a window other than the selected one—even if the buffer is already displayed in the selected window.

Like `switch-to-buffer`, this function updates the buffer list unless *norecord* is non-`nil`.

## 17.12 Choosing a Window for Display

The command `display-buffer` flexibly chooses a window for display, and displays a specified buffer in that window. It can be called interactively, via the key binding `C-x 4 C-o`. It is also used as a subroutine by many functions and commands, including `switch-to-buffer` and `pop-to-buffer` (see Section 17.11 [Switching Buffers], page 311).

This command performs several complex steps to find a window to display in. These steps are described by means of *display actions*, which have the form `(`*`function . alist`*`)`. Here, *function* is either a function or a list of functions, which we refer to as *action functions*; *alist* is an association list, which we refer to as *action alists*.

An action function accepts two arguments: the buffer to display and an action alist. It attempts to display the buffer in some window, picking or creating a window according to its own criteria. If successful, it returns the window; otherwise, it returns `nil`. See Section 17.13 [Display Action Functions], page 314, for a list of predefined action functions.

`display-buffer` works by combining display actions from several sources, and calling the action functions in turn, until one of them manages to display the buffer and returns a non-`nil` value.

`display-buffer` *buffer-or-name* **&optional** *action frame* [Command]

This command makes *buffer-or-name* appear in some window, without selecting the window or making the buffer current. The argument *buffer-or-name* must be a buffer or the name of an existing buffer. The return value is the window chosen to display the buffer.

The optional argument *action*, if non-`nil`, should normally be a display action (described above). `display-buffer` builds a list of action functions and an action alist, by consolidating display actions from the following sources (in order):

- The variable `display-buffer-overriding-action`.
- The user option `display-buffer-alist`.
- The *action* argument.
- The user option `display-buffer-base-action`.
- The constant `display-buffer-fallback-action`.

Each action function is called in turn, passing the buffer as the first argument and the combined action alist as the second argument, until one of the functions returns non-`nil`.

The argument *action* can also have a non-`nil`, non-list value. This has the special meaning that the buffer should be displayed in a window other than the selected one, even if the selected window is already displaying it. If called interactively with a prefix argument, *action* is `t`.

The optional argument *frame*, if non-`nil`, specifies which frames to check when deciding whether the buffer is already displayed. It is equivalent to adding an element `(reusable-frames . frame)` to the action alist of *action*. See Section 17.13 [Display Action Functions], page 314.

`display-buffer-overriding-action` [Variable]
The value of this variable should be a display action, which is treated with the highest priority by `display-buffer`. The default value is empty, i.e., `(nil . nil)`.

`display-buffer-alist` [User Option]
The value of this option is an alist mapping conditions to display actions. Each condition may be either a regular expression matching a buffer name or a function that takes two arguments: a buffer name and the *action* argument passed to `display-buffer`. If the name of the buffer passed to `display-buffer` either matches a regular expression in this alist or the function specified by a condition returns non-`nil`, then `display-buffer` uses the corresponding display action to display the buffer.

`display-buffer-base-action` [User Option]
The value of this option should be a display action. This option can be used to define a "standard" display action for calls to `display-buffer`.

`display-buffer-fallback-action` [Constant]
This display action specifies the fallback behavior for `display-buffer` if no other display actions are given.

## 17.13 Action Functions for `display-buffer`

The following basic action functions are defined in Emacs. Each of these functions takes two arguments: *buffer*, the buffer to display, and *alist*, an action alist. Each action function returns the window if it succeeds, and `nil` if it fails.

`display-buffer-same-window` *buffer alist* [Function]
This function tries to display *buffer* in the selected window. It fails if the selected window is a minibuffer window or is dedicated to another buffer (see Section 17.16 [Dedicated Windows], page 321). It also fails if *alist* has a non-`nil` `inhibit-same-window` entry.

`display-buffer-reuse-window` *buffer alist* [Function]
This function tries to "display" *buffer* by finding a window that is already displaying it.

If *alist* has a non-`nil` `inhibit-same-window` entry, the selected window is not eligible for reuse. If *alist* contains a `reusable-frames` entry, its value determines which frames to search for a reusable window:

- `nil` means consider windows on the selected frame. (Actually, the last non-minibuffer frame.)
- `t` means consider windows on all frames.
- `visible` means consider windows on all visible frames.
- 0 means consider windows on all visible or iconified frames.

- A frame means consider windows on that frame only.

If *alist* contains no `reusable-frames` entry, this function normally searches just the selected frame; however, if the variable `pop-up-frames` is non-`nil`, it searches all frames on the current terminal. See Section 17.14 [Choosing Window Options], page 317.

If this function chooses a window on another frame, it makes that frame visible and, unless *alist* contains an `inhibit-switch-frame` entry (see Section 17.14 [Choosing Window Options], page 317), raises that frame if necessary.

`display-buffer-pop-up-frame` *buffer alist* [Function]

This function creates a new frame, and displays the buffer in that frame's window. It actually performs the frame creation by calling the function specified in `pop-up-frame-function` (see Section 17.14 [Choosing Window Options], page 317). If *alist* contains a `pop-up-frame-parameters` entry, the associated value is added to the newly created frame's parameters.

`display-buffer-pop-up-window` *buffer alist* [Function]

This function tries to display *buffer* by splitting the largest or least recently-used window (typically one on the selected frame). It actually performs the split by calling the function specified in `split-window-preferred-function` (see Section 17.14 [Choosing Window Options], page 317).

The size of the new window can be adjusted by supplying `window-height` and `window-width` entries in *alist*. To adjust the window's height, use an entry whose CAR is `window-height` and whose CDR is one of:

- `nil` means to leave the height of the new window alone.
- A number specifies the desired height of the new window. An integer number specifies the number of lines of the window. A floating point number gives the fraction of the window's height with respect to the height of the frame's root window.
- If the CDR specifies a function, that function is called with one argument: the new window. The function is supposed to adjust the height of the window; its return value is ignored. Suitable functions are `shrink-window-if-larger-than-buffer` and `fit-window-to-buffer`, see Section 17.4 [Resizing Windows], page 295.

To adjust the window's width, use an entry whose CAR is `window-width` and whose CDR is one of:

- `nil` means to leave the width of the new window alone.
- A number specifies the desired width of the new window. An integer number specifies the number of columns of the window. A floating point number gives the fraction of the window's width with respect to the width of the frame's root window.
- If the CDR specifies a function, that function is called with one argument: the new window. The function is supposed to adjust the width of the window; its return value is ignored.

This function can fail if no window splitting can be performed for some reason (e.g., if the selected frame has an `unsplittable` frame parameter; see Section 18.3.3.5 [Buffer Parameters], page 350).

`display-buffer-below-selected` *buffer alist* [Function]
This function tries to display *buffer* in a window below the selected window. This means to either split the selected window or use the window below the selected one. If it does create a new window, it will also adjust its size provided *alist* contains a suitable `window-height` or `window-width` entry, see above.

`display-buffer-in-previous-window` *buffer alist* [Function]
This function tries to display *buffer* in a window previously showing it. If *alist* has a non-`nil` `inhibit-same-window` entry, the selected window is not eligible for reuse. If *alist* contains a `reusable-frames` entry, its value determines which frames to search for a suitable window as with `display-buffer-reuse-window`.

If *alist* has a `previous-window` entry, the window specified by that entry will override any other window found by the methods above, even if that window never showed *buffer* before.

`display-buffer-use-some-window` *buffer alist* [Function]
This function tries to display *buffer* by choosing an existing window and displaying the buffer in that window. It can fail if all windows are dedicated to another buffer (see Section 17.16 [Dedicated Windows], page 321).

To illustrate the use of action functions, consider the following example.

```
(display-buffer
 (get-buffer-create "*foo*")
 '((display-buffer-reuse-window
    display-buffer-pop-up-window
    display-buffer-pop-up-frame)
   (reusable-frames . 0)
   (window-height . 10) (window-width . 40)))
```

Evaluating the form above will cause `display-buffer` to proceed as follows: If a buffer called *foo* already appears on a visible or iconified frame, it will reuse its window. Otherwise, it will try to pop up a new window or, if that is impossible, a new frame and show the buffer there. If all these steps fail, it will proceed using whatever `display-buffer-base-action` and `display-buffer-fallback-action` prescribe.

Furthermore, `display-buffer` will try to adjust a reused window (provided *foo* was put by `display-buffer` there before) or a popped-up window as follows: If the window is part of a vertical combination, it will set its height to ten lines. Note that if, instead of the number "10", we specified the function `fit-window-to-buffer`, `display-buffer` would come up with a one-line window to fit the empty buffer. If the window is part of a horizontal combination, it sets its width to 40 columns. Whether a new window is vertically or horizontally combined depends on the shape of the window split and the values of `split-window-preferred-function`, `split-height-threshold` and `split-width-threshold` (see Section 17.14 [Choosing Window Options], page 317).

Now suppose we combine this call with a preexisting setup for 'display-buffer-alist' as follows.

```
(let ((display-buffer-alist
       (cons
        '("\\*foo\\*"
          (display-buffer-reuse-window display-buffer-below-selected)
          (reusable-frames)
          (window-height . 5))
        display-buffer-alist)))
  (display-buffer
   (get-buffer-create "*foo*")
   '((display-buffer-reuse-window
      display-buffer-pop-up-window
      display-buffer-pop-up-frame)
     (reusable-frames . 0)
     (window-height . 10) (window-width . 40))))
```

This form will have `display-buffer` first try reusing a window that shows *foo* on the selected frame. If there's no such window, it will try to split the selected window or, if that is impossible, use the window below the selected window.

If there's no window below the selected one, or the window below the selected one is dedicated to its buffer, `display-buffer` will proceed as described in the previous example. Note, however, that when it tries to adjust the height of any reused or popped-up window, it will in any case try to set its number of lines to "5" since that value overrides the corresponding specification in the *action* argument of `display-buffer`.

## 17.14 Additional Options for Displaying Buffers

The behavior of the standard display actions of `display-buffer` (see Section 17.12 [Choosing Window], page 313) can be modified by a variety of user options.

`pop-up-windows` [User Option]

If the value of this variable is non-`nil`, `display-buffer` is allowed to split an existing window to make a new window for displaying in. This is the default.

This variable is provided mainly for backward compatibility. It is obeyed by `display-buffer` via a special mechanism in `display-buffer-fallback-action`, which only calls the action function `display-buffer-pop-up-window` (see Section 17.13 [Display Action Functions], page 314) when the value is `nil`. It is not consulted by `display-buffer-pop-up-window` itself, which the user may specify directly in `display-buffer-alist` etc.

`split-window-preferred-function` [User Option]

This variable specifies a function for splitting a window, in order to make a new window for displaying a buffer. It is used by the `display-buffer-pop-up-window` action function to actually split the window (see Section 17.13 [Display Action Functions], page 314).

The default value is `split-window-sensibly`, which is documented below. The value must be a function that takes one argument, a window, and return either a new window (which will be used to display the desired buffer) or `nil` (which means the splitting failed).

`split-window-sensibly` *window* [Function]
This function tries to split *window*, and return the newly created window. If *window* cannot be split, it returns `nil`.

This function obeys the usual rules that determine when a window may be split (see Section 17.5 [Splitting Windows], page 297). It first tries to split by placing the new window below, subject to the restriction imposed by `split-height-threshold` (see below), in addition to any other restrictions. If that fails, it tries to split by placing the new window to the right, subject to `split-width-threshold` (see below). If that fails, and the window is the only window on its frame, this function again tries to split and place the new window below, disregarding `split-height-threshold`. If this fails as well, this function gives up and returns `nil`.

`split-height-threshold` [User Option]
This variable, used by `split-window-sensibly`, specifies whether to split the window placing the new window below. If it is an integer, that means to split only if the original window has at least that many lines. If it is `nil`, that means not to split this way.

`split-width-threshold` [User Option]
This variable, used by `split-window-sensibly`, specifies whether to split the window placing the new window to the right. If the value is an integer, that means to split only if the original window has at least that many columns. If the value is `nil`, that means not to split this way.

`pop-up-frames` [User Option]
If the value of this variable is non-`nil`, that means `display-buffer` may display buffers by making new frames. The default is `nil`.

A non-`nil` value also means that when `display-buffer` is looking for a window already displaying *buffer-or-name*, it can search any visible or iconified frame, not just the selected frame.

This variable is provided mainly for backward compatibility. It is obeyed by `display-buffer` via a special mechanism in `display-buffer-fallback-action`, which calls the action function `display-buffer-pop-up-frame` (see Section 17.13 [Display Action Functions], page 314) if the value is non-`nil`. (This is done before attempting to split a window.) This variable is not consulted by `display-buffer-pop-up-frame` itself, which the user may specify directly in `display-buffer-alist` etc.

`pop-up-frame-function` [User Option]
This variable specifies a function for creating a new frame, in order to make a new window for displaying a buffer. It is used by the `display-buffer-pop-up-frame` action function (see Section 17.13 [Display Action Functions], page 314).

The value should be a function that takes no arguments and returns a frame, or `nil` if no frame could be created. The default value is a function that creates a frame using the parameters specified by `pop-up-frame-alist` (see below).

`pop-up-frame-alist` [User Option]
This variable holds an alist of frame parameters (see Section 18.3 [Frame Parameters], page 345), which is used by the default function in `pop-up-frame-function` to make a new frame. The default is `nil`.

`same-window-buffer-names` [User Option]
: A list of buffer names for buffers that should be displayed in the selected window. If a buffer's name is in this list, `display-buffer` handles the buffer by showing it in the selected window.

`same-window-regexps` [User Option]
: A list of regular expressions that specify buffers that should be displayed in the selected window. If the buffer's name matches any of the regular expressions in this list, `display-buffer` handles the buffer by showing it in the selected window.

`same-window-p` *buffer-name* [Function]
: This function returns `t` if displaying a buffer named *buffer-name* with `display-buffer` would put it in the selected window.

## 17.15 Window History

Each window remembers in a list the buffers it has previously displayed, and the order in which these buffers were removed from it. This history is used, for example, by `replace-buffer-in-windows` (see Section 17.10 [Buffers and Windows], page 309). The list is automatically maintained by Emacs, but you can use the following functions to explicitly inspect or alter it:

`window-prev-buffers` **&optional** *window* [Function]
: This function returns a list specifying the previous contents of *window*. The optional argument *window* should be a live window and defaults to the selected one.

  Each list element has the form `(buffer window-start window-pos)`, where *buffer* is a buffer previously shown in the window, *window-start* is the window start position when that buffer was last shown, and *window-pos* is the point position when that buffer was last shown in *window*.

  The list is ordered so that earlier elements correspond to more recently-shown buffers, and the first element usually corresponds to the buffer most recently removed from the window.

`set-window-prev-buffers` *window prev-buffers* [Function]
: This function sets *window*'s previous buffers to the value of *prev-buffers*. The argument *window* must be a live window and defaults to the selected one. The argument *prev-buffers* should be a list of the same form as that returned by `window-prev-buffers`.

In addition, each buffer maintains a list of *next buffers*, which is a list of buffers re-shown by `switch-to-prev-buffer` (see below). This list is mainly used by `switch-to-prev-buffer` and `switch-to-next-buffer` for choosing buffers to switch to.

`window-next-buffers` **&optional** *window* [Function]
: This function returns the list of buffers recently re-shown in *window* via `switch-to-prev-buffer`. The *window* argument must denote a live window or `nil` (meaning the selected window).

`set-window-next-buffers` *window next-buffers* [Function]
: This function sets the next buffer list of *window* to *next-buffers*. The *window* argument should be a live window or `nil` (meaning the selected window). The argument *next-buffers* should be a list of buffers.

The following commands can be used to cycle through the global buffer list, much like `bury-buffer` and `unbury-buffer`. However, they cycle according to the specified window's history list, rather than the global buffer list. In addition, they restore window-specific window start and point positions, and may show a buffer even if it is already shown in another window. The `switch-to-prev-buffer` command, in particular, is used by `replace-buffer-in-windows`, `bury-buffer` and `quit-window` to find a replacement buffer for a window.

`switch-to-prev-buffer` **&optional** *window bury-or-kill* [Command]
: This command displays the previous buffer in *window*. The argument *window* should be a live window or `nil` (meaning the selected window). If the optional argument *bury-or-kill* is non-`nil`, this means that the buffer currently shown in *window* is about to be buried or killed and consequently should not be switched to in future invocations of this command.

  The previous buffer is usually the buffer shown before the buffer currently shown in *window*. However, a buffer that has been buried or killed, or has been already shown by a recent invocation of `switch-to-prev-buffer`, does not qualify as previous buffer.

  If repeated invocations of this command have already shown all buffers previously shown in *window*, further invocations will show buffers from the buffer list of the frame *window* appears on (see Section 16.8 [The Buffer List], page 281), trying to skip buffers that are already shown in another window on that frame.

`switch-to-next-buffer` **&optional** *window* [Command]
: This command switches to the next buffer in *window*, thus undoing the effect of the last `switch-to-prev-buffer` command in *window*. The argument *window* must be a live window and defaults to the selected one.

  If there is no recent invocation of `switch-to-prev-buffer` that can be undone, this function tries to show a buffer from the buffer list of the frame *window* appears on (see Section 16.8 [The Buffer List], page 281).

By default `switch-to-prev-buffer` and `switch-to-next-buffer` can switch to a buffer that is already shown in another window on the same frame. The following option can be used to override this behavior.

`switch-to-visible-buffer` [User Option]
: If this variable is non-`nil`, `switch-to-prev-buffer` and `switch-to-next-buffer` may switch to a buffer that is already visible on the same frame, provided the buffer was shown in the relevant window before. If it is `nil`, `switch-to-prev-buffer` and `switch-to-next-buffer` always try to avoid switching to a buffer that is already visible in another window on the same frame.

## 17.16 Dedicated Windows

Functions for displaying a buffer can be told to not use specific windows by marking these windows as *dedicated* to their buffers. `display-buffer` (see Section 17.12 [Choosing Window], page 313) never uses a dedicated window for displaying another buffer in it. `get-lru-window` and `get-largest-window` (see Section 17.9 [Cyclic Window Ordering], page 307) do not consider dedicated windows as candidates when their *dedicated* argument is non-`nil`. The behavior of `set-window-buffer` (see Section 17.10 [Buffers and Windows], page 309) with respect to dedicated windows is slightly different, see below.

Functions supposed to remove a buffer from a window or a window from a frame can behave specially when a window they operate on is dedicated. We will distinguish three basic cases, namely where (1) the window is not the only window on its frame, (2) the window is the only window on its frame but there are other frames on the same terminal left, and (3) the window is the only window on the only frame on the same terminal.

In particular, `delete-windows-on` (see Section 17.6 [Deleting Windows], page 300) handles case (2) by deleting the associated frame and case (3) by showing another buffer in that frame's only window. The function `replace-buffer-in-windows` (see Section 17.10 [Buffers and Windows], page 309) which is called when a buffer gets killed, deletes the window in case (1) and behaves like `delete-windows-on` otherwise.

When `bury-buffer` (see Section 16.8 [The Buffer List], page 281) operates on the selected window (which shows the buffer that shall be buried), it handles case (2) by calling `frame-auto-hide-function` (see Section 17.17 [Quitting Windows], page 321) to deal with the selected frame. The other two cases are handled as with `replace-buffer-in-windows`.

`window-dedicated-p` **&optional** *window* [Function]
: This function returns non-`nil` if *window* is dedicated to its buffer and `nil` otherwise. More precisely, the return value is the value assigned by the last call of `set-window-dedicated-p` for *window*, or `nil` if that function was never called with *window* as its argument. The default for *window* is the selected window.

`set-window-dedicated-p` *window flag* [Function]
: This function marks *window* as dedicated to its buffer if *flag* is non-`nil`, and non-dedicated otherwise.

  As a special case, if *flag* is `t`, *window* becomes *strongly* dedicated to its buffer. `set-window-buffer` signals an error when the window it acts upon is strongly dedicated to its buffer and does not already display the buffer it is asked to display. Other functions do not treat `t` differently from any non-`nil` value.

## 17.17 Quitting Windows

When you want to get rid of a window used for displaying a buffer, you can call `delete-window` or `delete-windows-on` (see Section 17.6 [Deleting Windows], page 300) to remove that window from its frame. If the buffer is shown on a separate frame, you might want to call `delete-frame` (see Section 18.6 [Deleting Frames], page 357) instead. If, on the other hand, a window has been reused for displaying the buffer, you might prefer showing the buffer previously shown in that window, by calling the function `switch-to-prev-buffer` (see Section 17.15 [Window History], page 319). Finally, you might want to either bury

(see Section 16.8 [The Buffer List], page 281) or kill (see Section 16.10 [Killing Buffers], page 284) the window's buffer.

The following command uses information on how the window for displaying the buffer was obtained in the first place, thus attempting to automate the above decisions for you.

`quit-window` **&optional** *kill window* [Command]
: This command quits *window* and buries its buffer. The argument *window* must be a live window and defaults to the selected one. With prefix argument *kill* non-`nil`, it kills the buffer instead of burying it. It calls the function `quit-restore-window` described next to deal with the window and its buffer.

`quit-restore-window` **&optional** *window bury-or-kill* [Function]
: This function tries to restore the state of *window* that existed before its buffer was displayed in it. The optional argument *window* must be a live window and defaults to the selected one.

  If *window* was created specially for displaying its buffer, this function deletes *window* provided its frame contains at least one other live window. If *window* is the only window on its frame and there are other frames on the frame's terminal, the value of the optional argument *bury-or-kill* determines how to proceed with the window. If *bury-or-kill* equals `kill`, the frame is deleted unconditionally. Otherwise, the fate of the frame is determined by calling `frame-auto-hide-function` (see below) with that frame as sole argument.

  Otherwise, this function tries to redisplay the buffer previously shown in *window*. It also tries to restore the window start (see Section 17.19 [Window Start and End], page 324) and point (see Section 17.18 [Window Point], page 323) positions of the previously shown buffer. If, in addition, *window*'s buffer was temporarily resized, this function will also try to restore the original height of *window*.

  The cases described so far require that the buffer shown in *window* is still the buffer displayed by the last buffer display function for this window. If another buffer has been shown in the meantime, or the buffer previously shown no longer exists, this function calls `switch-to-prev-buffer` (see Section 17.15 [Window History], page 319) to show some other buffer instead.

  The optional argument *bury-or-kill* specifes how to deal with *window*'s buffer. The following values are handled:

  `nil` This means to not deal with the buffer in any particular way. As a consequence, if *window* is not deleted, invoking `switch-to-prev-buffer` will usually show the buffer again.

  `append` This means that if *window* is not deleted, its buffer is moved to the end of *window*'s list of previous buffers, so it's less likely that a future invocation of `switch-to-prev-buffer` will switch to it. Also, it moves the buffer to the end of the frame's buffer list.

  `bury` This means that if *window* is not deleted, its buffer is removed from *window*'s list of previous buffers. Also, it moves the buffer to the end of the frame's buffer list. This value provides the most reliable remedy to not have `switch-to-prev-buffer` switch to this buffer again without killing the buffer.

`kill` This means to kill *window*'s buffer.

`quit-restore-window` bases its decisions on information stored in *window*'s `quit-restore` window parameter (see Section 17.25 [Window Parameters], page 337), and resets that parameter to `nil` after it's done.

The following option specifies how to deal with a frame containing just one window that should be either quit, or whose buffer should be buried.

`frame-auto-hide-function` [User Option]

The function specified by this option is called to automatically hide frames. This function is called with one argument—a frame.

The function specified here is called by `bury-buffer` (see Section 16.8 [The Buffer List], page 281) when the selected window is dedicated and shows the buffer to bury. It is also called by `quit-restore-window` (see above) when the frame of the window to quit has been specially created for displaying that window's buffer and the buffer is not killed.

The default is to call `iconify-frame` (see Section 18.10 [Visibility of Frames], page 360). Alternatively, you may specify either `delete-frame` (see Section 18.6 [Deleting Frames], page 357) to remove the frame from its display, `ignore` to leave the frame unchanged, or any other function that can take a frame as its sole argument.

Note that the function specified by this option is called only if the specified frame contains just one live window and there is at least one other frame on the same terminal.

## 17.18 Windows and Point

Each window has its own value of point (see Section 1.1 [Point], page 6), independent of the value of point in other windows displaying the same buffer. This makes it useful to have multiple windows showing one buffer.

- The window point is established when a window is first created; it is initialized from the buffer's point, or from the window point of another window opened on the buffer if such a window exists.
- Selecting a window sets the value of point in its buffer from the window's value of point. Conversely, deselecting a window sets the window's value of point from that of the buffer. Thus, when you switch between windows that display a given buffer, the point value for the selected window is in effect in the buffer, while the point values for the other windows are stored in those windows.
- As long as the selected window displays the current buffer, the window's point and the buffer's point always move together; they remain equal.

As far as the user is concerned, point is where the cursor is, and when the user switches to another buffer, the cursor jumps to the position of point in that buffer.

`window-point` **&optional** *window* [Function]

This function returns the current position of point in *window*. For a nonselected window, this is the value point would have (in that window's buffer) if that window were selected. The default for *window* is the selected window.

When *window* is the selected window, the value returned is the value of point in that window's buffer. Strictly speaking, it would be more correct to return the "top-level" value of point, outside of any `save-excursion` forms. But that value is hard to find.

`set-window-point` *window position* [Function]
This function positions point in *window* at position *position* in *window*'s buffer. It returns *position*.

If *window* is selected, this simply does `goto-char` in *window*'s buffer.

`window-point-insertion-type` [Variable]
This variable specifies the marker insertion type (see ⟨undefined⟩ [Marker Insertion Types], page ⟨undefined⟩) of `window-point`. The default is `nil`, so `window-point` will stay behind text inserted there.

## 17.19 The Window Start and End Positions

Each window maintains a marker used to keep track of a buffer position that specifies where in the buffer display should start. This position is called the *display-start* position of the window (or just the *start*). The character after this position is the one that appears at the upper left corner of the window. It is usually, but not inevitably, at the beginning of a text line.

After switching windows or buffers, and in some other cases, if the window start is in the middle of a line, Emacs adjusts the window start to the start of a line. This prevents certain operations from leaving the window start at a meaningless point within a line. This feature may interfere with testing some Lisp code by executing it using the commands of Lisp mode, because they trigger this readjustment. To test such code, put it into a command and bind the command to a key.

`window-start` **&optional** *window* [Function]
This function returns the display-start position of window *window*. If *window* is `nil`, the selected window is used.

When you create a window, or display a different buffer in it, the display-start position is set to a display-start position recently used for the same buffer, or to `point-min` if the buffer doesn't have any.

Redisplay updates the window-start position (if you have not specified it explicitly since the previous redisplay)—to make sure point appears on the screen. Nothing except redisplay automatically changes the window-start position; if you move point, do not expect the window-start position to change in response until after the next redisplay.

`window-end` **&optional** *window update* [Function]
This function returns the position where display of its buffer ends in *window*. The default for *window* is the selected window.

Simply changing the buffer text or moving point does not update the value that `window-end` returns. The value is updated only when Emacs redisplays and redisplay completes without being preempted.

If the last redisplay of *window* was preempted, and did not finish, Emacs does not know the position of the end of display in that window. In that case, this function returns `nil`.

If *update* is non-`nil`, `window-end` always returns an up-to-date value for where display ends, based on the current `window-start` value. If a previously saved value of that position is still valid, `window-end` returns that value; otherwise it computes the correct value by scanning the buffer text.

Even if *update* is non-`nil`, `window-end` does not attempt to scroll the display if point has moved off the screen, the way real redisplay would do. It does not alter the `window-start` value. In effect, it reports where the displayed text will end if scrolling is not required.

`set-window-start` *window position* **&optional** *noforce* [Function]

This function sets the display-start position of *window* to *position* in *window*'s buffer. It returns *position*.

The display routines insist that the position of point be visible when a buffer is displayed. Normally, they change the display-start position (that is, scroll the window) whenever necessary to make point visible. However, if you specify the start position with this function using `nil` for *noforce*, it means you want display to start at *position* even if that would put the location of point off the screen. If this does place point off screen, the display routines move point to the left margin on the middle line in the window.

For example, if point is 1 and you set the start of the window to 37, the start of the next line, point will be "above" the top of the window. The display routines will automatically move point if it is still 1 when redisplay occurs. Here is an example:

```
;; Here is what 'foo' looks like before executing
;;    the set-window-start expression.

---------- Buffer: foo ----------
*This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (save-excursion
   (goto-char 1)
   (forward-line 1)
   (point)))
⇒ 37
```

```
;; Here is what 'foo' looks like after executing
;;   the set-window-start expression.
---------- Buffer: foo ----------
2
3
⋆4
5
6
---------- Buffer: foo ----------
```

If *noforce* is non-`nil`, and *position* would place point off screen at the next redisplay, then redisplay computes a new window-start position that works well with point, and thus *position* is not used.

`pos-visible-in-window-p` **&optional** *position window partially* [Function]

This function returns non-`nil` if *position* is within the range of text currently visible on the screen in *window*. It returns `nil` if *position* is scrolled vertically out of view. Locations that are partially obscured are not considered visible unless *partially* is non-`nil`. The argument *position* defaults to the current position of point in *window*; *window*, to the selected window. If *position* is `t`, that means to check the last visible position in *window*.

This function considers only vertical scrolling. If *position* is out of view only because *window* has been scrolled horizontally, `pos-visible-in-window-p` returns non-`nil` anyway. See Section 17.22 [Horizontal Scrolling], page 331.

If *position* is visible, `pos-visible-in-window-p` returns `t` if *partially* is `nil`; if *partially* is non-`nil`, and the character following *position* is fully visible, it returns a list of the form `(x y)`, where *x* and *y* are the pixel coordinates relative to the top left corner of the window; otherwise it returns an extended list of the form `(x y rtop rbot rowh vpos)`, where *rtop* and *rbot* specify the number of off-window pixels at the top and bottom of the row at *position*, *rowh* specifies the visible height of that row, and *vpos* specifies the vertical position (zero-based row number) of that row.

Here is an example:

```
;; If point is off the screen now, recenter it now.
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

`window-line-height` **&optional** *line window* [Function]

This function returns the height of text line *line* in *window*. If *line* is one of `header-line` or `mode-line`, `window-line-height` returns information about the corresponding line of the window. Otherwise, *line* is a text line number starting from 0. A negative number counts from the end of the window. The default for *line* is the current line in *window*; the default for *window* is the selected window.

If the display is not up to date, `window-line-height` returns `nil`. In that case, `pos-visible-in-window-p` may be used to obtain related information.

If there is no line corresponding to the specified *line*, `window-line-height` returns `nil`. Otherwise, it returns a list `(height vpos ypos offbot)`, where *height* is the

height in pixels of the visible part of the line, *vpos* and *ypos* are the vertical position in lines and pixels of the line relative to the top of the first text line, and *offbot* is the number of off-window pixels at the bottom of the text line. If there are off-window pixels at the top of the (first) text line, *ypos* is negative.

## 17.20 Textual Scrolling

*Textual scrolling* means moving the text up or down through a window. It works by changing the window's display-start location. It may also change the value of `window-point` to keep point on the screen (see Section 17.18 [Window Point], page 323).

The basic textual scrolling functions are `scroll-up` (which scrolls forward) and `scroll-down` (which scrolls backward). In these function names, "up" and "down" refer to the direction of motion of the buffer text relative to the window. Imagine that the text is written on a long roll of paper and that the scrolling commands move the paper up and down. Thus, if you are looking at the middle of a buffer and repeatedly call `scroll-down`, you will eventually see the beginning of the buffer.

Unfortunately, this sometimes causes confusion, because some people tend to think in terms of the opposite convention: they imagine the window moving over text that remains in place, so that "down" commands take you to the end of the buffer. This convention is consistent with fact that such a command is bound to a key named PAGEDOWN on modern keyboards.

Textual scrolling functions (aside from `scroll-other-window`) have unpredictable results if the current buffer is not the one displayed in the selected window. See Section 16.2 [Current Buffer], page 272.

If the window contains a row taller than the height of the window (for example in the presence of a large image), the scroll functions will adjust the window's vertical scroll position to scroll the partially visible row. Lisp callers can disable this feature by binding the variable `auto-window-vscroll` to `nil` (see Section 17.21 [Vertical Scrolling], page 330).

`scroll-up` **&optional** *count* [Command]

This function scrolls forward by *count* lines in the selected window.

If *count* is negative, it scrolls backward instead. If *count* is `nil` (or omitted), the distance scrolled is `next-screen-context-lines` lines less than the height of the window's text area.

If the selected window cannot be scrolled any further, this function signals an error. Otherwise, it returns `nil`.

`scroll-down` **&optional** *count* [Command]

This function scrolls backward by *count* lines in the selected window.

If *count* is negative, it scrolls forward instead. In other respects, it behaves the same way as `scroll-up` does.

`scroll-up-command` **&optional** *count* [Command]

This behaves like `scroll-up`, except that if the selected window cannot be scrolled any further and the value of the variable `scroll-error-top-bottom` is `t`, it tries to move to the end of the buffer instead. If point is already there, it signals an error.

`scroll-down-command` **&optional** *count* [Command]
: This behaves like `scroll-down`, except that if the selected window cannot be scrolled any further and the value of the variable `scroll-error-top-bottom` is `t`, it tries to move to the beginning of the buffer instead. If point is already there, it signals an error.

`scroll-other-window` **&optional** *count* [Command]
: This function scrolls the text in another window upward *count* lines. Negative values of *count*, or `nil`, are handled as in `scroll-up`.

  You can specify which buffer to scroll by setting the variable `other-window-scroll-buffer` to a buffer. If that buffer isn't already displayed, `scroll-other-window` displays it in some window.

  When the selected window is the minibuffer, the next window is normally the leftmost one immediately above it. You can specify a different window to scroll, when the minibuffer is selected, by setting the variable `minibuffer-scroll-window`. This variable has no effect when any other window is selected. When it is non-`nil` and the minibuffer is selected, it takes precedence over `other-window-scroll-buffer`. See ⟨undefined⟩ [Definition of minibuffer-scroll-window], page ⟨undefined⟩.

  When the minibuffer is active, it is the next window if the selected window is the one at the bottom right corner. In this case, `scroll-other-window` attempts to scroll the minibuffer. If the minibuffer contains just one line, it has nowhere to scroll to, so the line reappears after the echo area momentarily displays the message '`End of buffer`'.

`other-window-scroll-buffer` [Variable]
: If this variable is non-`nil`, it tells `scroll-other-window` which buffer's window to scroll.

`scroll-margin` [User Option]
: This option specifies the size of the scroll margin—a minimum number of lines between point and the top or bottom of a window. Whenever point gets within this many lines of the top or bottom of the window, redisplay scrolls the text automatically (if possible) to move point out of the margin, closer to the center of the window.

`scroll-conservatively` [User Option]
: This variable controls how scrolling is done automatically when point moves off the screen (or into the scroll margin). If the value is a positive integer *n*, then redisplay scrolls the text up to *n* lines in either direction, if that will bring point back into proper view. This behavior is called *conservative scrolling*. Otherwise, scrolling happens in the usual way, under the control of other variables such as `scroll-up-aggressively` and `scroll-down-aggressively`.

  The default value is zero, which means that conservative scrolling never happens.

`scroll-down-aggressively` [User Option]
: The value of this variable should be either `nil` or a fraction *f* between 0 and 1. If it is a fraction, that specifies where on the screen to put point when scrolling down. More precisely, when a window scrolls down because point is above the window start,

the new start position is chosen to put point *f* part of the window height from the top. The larger *f*, the more aggressive the scrolling.

A value of `nil` is equivalent to .5, since its effect is to center point. This variable automatically becomes buffer-local when set in any fashion.

`scroll-up-aggressively` [User Option]
Likewise, for scrolling up. The value, *f*, specifies how far point should be placed from the bottom of the window; thus, as with `scroll-up-aggressively`, a larger value scrolls more aggressively.

`scroll-step` [User Option]
This variable is an older variant of `scroll-conservatively`. The difference is that if its value is *n*, that permits scrolling only by precisely *n* lines, not a smaller number. This feature does not work with `scroll-margin`. The default value is zero.

`scroll-preserve-screen-position` [User Option]
If this option is `t`, whenever a scrolling command moves point off-window, Emacs tries to adjust point to keep the cursor at its old vertical position in the window, rather than the window edge.

If the value is non-`nil` and not `t`, Emacs adjusts point to keep the cursor at the same vertical position, even if the scrolling command didn't move point off-window.

This option affects all scroll commands that have a non-`nil` `scroll-command` symbol property.

`next-screen-context-lines` [User Option]
The value of this variable is the number of lines of continuity to retain when scrolling by full screens. For example, `scroll-up` with an argument of `nil` scrolls so that this many lines at the bottom of the window appear instead at the top. The default value is 2.

`scroll-error-top-bottom` [User Option]
If this option is `nil` (the default), `scroll-up-command` and `scroll-down-command` simply signal an error when no more scrolling is possible.

If the value is `t`, these commands instead move point to the beginning or end of the buffer (depending on scrolling direction); only if point is already on that position do they signal an error.

`recenter` **&optional** *count* [Command]
This function scrolls the text in the selected window so that point is displayed at a specified vertical position within the window. It does not "move point" with respect to the text.

If *count* is a non-negative number, that puts the line containing point *count* lines down from the top of the window. If *count* is a negative number, then it counts upward from the bottom of the window, so that −1 stands for the last usable line in the window.

If *count* is `nil` (or a non-`nil` list), `recenter` puts the line containing point in the middle of the window. If *count* is `nil`, this function may redraw the frame, according to the value of `recenter-redisplay`.

When `recenter` is called interactively, *count* is the raw prefix argument. Thus, typing `C-u` as the prefix sets the *count* to a non-`nil` list, while typing `C-u 4` sets *count* to 4, which positions the current line four lines from the top.

With an argument of zero, `recenter` positions the current line at the top of the window. The command `recenter-top-bottom` offers a more convenient way to achieve this.

`recenter-redisplay` [User Option]
If this variable is non-`nil`, calling `recenter` with a `nil` argument redraws the frame. The default value is `tty`, which means only redraw the frame if it is a tty frame.

`recenter-top-bottom` **&optional** *count* [Command]
This command, which is the default binding for `C-l`, acts like `recenter`, except if called with no argument. In that case, successive calls place point according to the cycling order defined by the variable `recenter-positions`.

`recenter-positions` [User Option]
This variable controls how `recenter-top-bottom` behaves when called with no argument. The default value is `(middle top bottom)`, which means that successive calls of `recenter-top-bottom` with no argument cycle between placing point at the middle, top, and bottom of the window.

## 17.21 Vertical Fractional Scrolling

*Vertical fractional scrolling* means shifting text in a window up or down by a specified multiple or fraction of a line. Each window has a *vertical scroll position*, which is a number, never less than zero. It specifies how far to raise the contents of the window. Raising the window contents generally makes all or part of some lines disappear off the top, and all or part of some other lines appear at the bottom. The usual value is zero.

The vertical scroll position is measured in units of the normal line height, which is the height of the default font. Thus, if the value is .5, that means the window contents are scrolled up half the normal line height. If it is 3.3, that means the window contents are scrolled up somewhat over three times the normal line height.

What fraction of a line the vertical scrolling covers, or how many lines, depends on what the lines contain. A value of .5 could scroll a line whose height is very short off the screen, while a value of 3.3 could scroll just part of the way through a tall line or an image.

`window-vscroll` **&optional** *window pixels-p* [Function]
This function returns the current vertical scroll position of *window*. The default for *window* is the selected window. If *pixels-p* is non-`nil`, the return value is measured in pixels, rather than in units of the normal line height.

```
(window-vscroll)
     ⇒ 0
```

`set-window-vscroll` *window lines* **&optional** *pixels-p* [Function]
This function sets *window*'s vertical scroll position to *lines*. If *window* is `nil`, the selected window is used. The argument *lines* should be zero or positive; if not, it is taken as zero.

The actual vertical scroll position must always correspond to an integral number of pixels, so the value you specify is rounded accordingly.

The return value is the result of this rounding.

```
(set-window-vscroll (selected-window) 1.2)
     ⇒ 1.13
```

If *pixels-p* is non-`nil`, *lines* specifies a number of pixels. In this case, the return value is *lines*.

`auto-window-vscroll` [Variable]

If this variable is non-`nil`, the `line-move`, `scroll-up`, and `scroll-down` functions will automatically modify the vertical scroll position to scroll through display rows that are taller than the height of the window, for example in the presence of large images.

## 17.22 Horizontal Scrolling

*Horizontal scrolling* means shifting the image in the window left or right by a specified multiple of the normal character width. Each window has a *horizontal scroll position*, which is a number, never less than zero. It specifies how far to shift the contents left. Shifting the window contents left generally makes all or part of some characters disappear off the left, and all or part of some other characters appear at the right. The usual value is zero.

The horizontal scroll position is measured in units of the normal character width, which is the width of space in the default font. Thus, if the value is 5, that means the window contents are scrolled left by 5 times the normal character width. How many characters actually disappear off to the left depends on their width, and could vary from line to line.

Because we read from side to side in the "inner loop", and from top to bottom in the "outer loop", the effect of horizontal scrolling is not like that of textual or vertical scrolling. Textual scrolling involves selection of a portion of text to display, and vertical scrolling moves the window contents contiguously; but horizontal scrolling causes part of *each line* to go off screen.

Usually, no horizontal scrolling is in effect; then the leftmost column is at the left edge of the window. In this state, scrolling to the right is meaningless, since there is no data to the left of the edge to be revealed by it; so this is not allowed. Scrolling to the left is allowed; it scrolls the first columns of text off the edge of the window and can reveal additional columns on the right that were truncated before. Once a window has a nonzero amount of leftward horizontal scrolling, you can scroll it back to the right, but only so far as to reduce the net horizontal scroll to zero. There is no limit to how far left you can scroll, but eventually all the text will disappear off the left edge.

If `auto-hscroll-mode` is set, redisplay automatically alters the horizontal scrolling of a window as necessary to ensure that point is always visible. However, you can still set the horizontal scrolling value explicitly. The value you specify serves as a lower bound for automatic scrolling, i.e., automatic scrolling will not scroll a window to a column less than the specified one.

`scroll-left` **&optional** *count set-minimum* [Command]
: This function scrolls the selected window *count* columns to the left (or to the right if *count* is negative). The default for *count* is the window width, minus 2.

  The return value is the total amount of leftward horizontal scrolling in effect after the change—just like the value returned by `window-hscroll` (below).

  Once you scroll a window as far right as it can go, back to its normal position where the total leftward scrolling is zero, attempts to scroll any farther right have no effect.

  If *set-minimum* is non-`nil`, the new scroll amount becomes the lower bound for automatic scrolling; that is, automatic scrolling will not scroll a window to a column less than the value returned by this function. Interactive calls pass non-`nil` for *set-minimum*.

`scroll-right` **&optional** *count set-minimum* [Command]
: This function scrolls the selected window *count* columns to the right (or to the left if *count* is negative). The default for *count* is the window width, minus 2. Aside from the direction of scrolling, this works just like `scroll-left`.

`window-hscroll` **&optional** *window* [Function]
: This function returns the total leftward horizontal scrolling of *window*—the number of columns by which the text in *window* is scrolled left past the left margin. The default for *window* is the selected window.

  The return value is never negative. It is zero when no horizontal scrolling has been done in *window* (which is usually the case).

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

`set-window-hscroll` *window columns* [Function]
: This function sets horizontal scrolling of *window*. The value of *columns* specifies the amount of scrolling, in terms of columns from the left margin. The argument *columns* should be zero or positive; if not, it is taken as zero. Fractional values of *columns* are not supported at present.

  Note that `set-window-hscroll` may appear not to work if you test it by evaluating a call with `M-:` in a simple way. What happens is that the function sets the horizontal scroll value and returns, but then redisplay adjusts the horizontal scrolling to make point visible, and this overrides what the function did. You can observe the function's effect if you call it while point is sufficiently far from the left margin that it will remain visible.

  The value returned is *columns*.

```
(set-window-hscroll (selected-window) 10)
     ⇒ 10
```

Here is how you can determine whether a given position *position* is off the screen due to horizontal scrolling:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

## 17.23 Coordinates and Windows

This section describes functions that report the position of a window. Most of these functions report positions relative to the window's frame. In this case, the coordinate origin '`(0,0)`' lies near the upper left corner of the frame. For technical reasons, on graphical displays the origin is not located at the exact corner of the graphical window as it appears on the screen. If Emacs is built with the GTK+ toolkit, the origin is at the upper left corner of the frame area used for displaying Emacs windows, below the title-bar, GTK+ menu bar, and tool bar (since these are drawn by the window manager and/or GTK+, not by Emacs). But if Emacs is not built with GTK+, the origin is at the upper left corner of the tool bar (since in this case Emacs itself draws the tool bar). In both cases, the X and Y coordinates increase rightward and downward respectively.

Except where noted, X and Y coordinates are reported in integer character units, i.e., numbers of lines and columns respectively. On a graphical display, each "line" and "column" corresponds to the height and width of a default character specified by the frame's default font.

`window-edges` **&optional** *window* [Function]
: This function returns a list of the edge coordinates of *window*. If *window* is omitted or `nil`, it defaults to the selected window.

  The return value has the form (*left top right bottom*). These list elements are, respectively, the X coordinate of the leftmost column occupied by the window, the Y coordinate of the topmost row, the X coordinate one column to the right of the rightmost column, and the Y coordinate one row down from the bottommost row.

  Note that these are the actual outer edges of the window, including any header line, mode line, scroll bar, fringes, and display margins. On a text terminal, if the window has a neighbor on its right, its right edge includes the separator line between the window and its neighbor.

`window-inside-edges` **&optional** *window* [Function]
: This function is similar to `window-edges`, but the returned edge values are for the text area of the window. They exclude any header line, mode line, scroll bar, fringes, display margins, and vertical separator.

`window-top-line` **&optional** *window* [Function]
: This function returns the Y coordinate of the topmost row of *window*, equivalent to the *top* entry in the list returned by `window-edges`.

`window-left-column` **&optional** *window* [Function]
: This function returns the X coordinate of the leftmost column of *window*, equivalent to the *left* entry in the list returned by `window-edges`.

The following functions can be used to relate a set of frame-relative coordinates to a window:

`window-at` *x y* **&optional** *frame* [Function]
: This function returns the live window at the frame-relative coordinates *x* and *y*, on frame *frame*. If there is no window at that position, the return value is `nil`. If *frame* is omitted or `nil`, it defaults to the selected frame.

`coordinates-in-window-p` *coordinates window* [Function]
: This function checks whether a window *window* occupies the frame-relative coordinates *coordinates*, and if so, which part of the window that is. *window* should be a live window. *coordinates* should be a cons cell of the form `(x . y)`, where *x* and *y* are frame-relative coordinates.

  If there is no window at the specified position, the return value is `nil` . Otherwise, the return value is one of the following:

  `(relx . rely)`
  : The coordinates are inside *window*. The numbers *relx* and *rely* are the equivalent window-relative coordinates for the specified position, counting from 0 at the top left corner of the window.

  `mode-line`
  : The coordinates are in the mode line of *window*.

  `header-line`
  : The coordinates are in the header line of *window*.

  `vertical-line`
  : The coordinates are in the vertical line between *window* and its neighbor to the right. This value occurs only if the window doesn't have a scroll bar; positions in a scroll bar are considered outside the window for these purposes.

  `left-fringe`
  `right-fringe`
  : The coordinates are in the left or right fringe of the window.

  `left-margin`
  `right-margin`
  : The coordinates are in the left or right margin of the window.

  `nil` The coordinates are not in any part of *window*.

  The function `coordinates-in-window-p` does not require a frame as argument because it always uses the frame that *window* is on.

The following functions return window positions in pixels, rather than character units. Though mostly useful on graphical displays, they can also be called on text terminals, where the screen area of each text character is taken to be "one pixel".

`window-pixel-edges` **&optional** *window* [Function]
: This function returns a list of pixel coordinates for the edges of *window*. If *window* is omitted or `nil`, it defaults to the selected window.

The return value has the form `(left top right bottom)`. The list elements are, respectively, the X pixel coordinate of the left window edge, the Y pixel coordinate of the top edge, one more than the X pixel coordinate of the right edge, and one more than the Y pixel coordinate of the bottom edge.

`window-inside-pixel-edges` **&optional** *window* [Function]
This function is like `window-pixel-edges`, except that it returns the pixel coordinates for the edges of the window's text area, rather than the pixel coordinates for the edges of the window itself. *window* must specify a live window.

The following functions return window positions in pixels, relative to the display screen rather than the frame:

`window-absolute-pixel-edges` **&optional** *window* [Function]
This function is like `window-pixel-edges`, except that it returns the edge pixel coordinates relative to the top left corner of the display screen.

`window-inside-absolute-pixel-edges` **&optional** *window* [Function]
This function is like `window-inside-pixel-edges`, except that it returns the edge pixel coordinates relative to the top left corner of the display screen. *window* must specify a live window.

## 17.24 Window Configurations

A *window configuration* records the entire layout of one frame—all windows, their sizes, which buffers they contain, how those buffers are scrolled, and their values of point and the mark; also their fringes, margins, and scroll bar settings. It also includes the value of `minibuffer-scroll-window`. As a special exception, the window configuration does not record the value of point in the selected window for the current buffer.

You can bring back an entire frame layout by restoring a previously saved window configuration. If you want to record the layout of all frames instead of just one, use a frame configuration instead of a window configuration. See Section 18.12 [Frame Configurations], page 362.

`current-window-configuration` **&optional** *frame* [Function]
This function returns a new object representing *frame*'s current window configuration. The default for *frame* is the selected frame. The variable `window-persistent-parameters` specifies which window parameters (if any) are saved by this function. See Section 17.25 [Window Parameters], page 337.

`set-window-configuration` *configuration* [Function]
This function restores the configuration of windows and buffers as specified by *configuration*, for the frame that *configuration* was created for.

The argument *configuration* must be a value that was previously returned by `current-window-configuration`. The configuration is restored in the frame from which *configuration* was made, whether that frame is selected or not. This always counts as a window size change and triggers execution of the `window-size-change-functions` (see Section 17.26 [Window Hooks], page 339),

because `set-window-configuration` doesn't know how to tell whether the new configuration actually differs from the old one.

If the frame from which *configuration* was saved is dead, all this function does is restore the three variables `window-min-height`, `window-min-width` and `minibuffer-scroll-window`. In this case, the function returns `nil`. Otherwise, it returns `t`.

Here is a way of using this function to get the same effect as `save-window-excursion`:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-below nil)
             ...)
    (set-window-configuration config)))
```

`save-window-excursion` *forms...* [Macro]

This macro records the window configuration of the selected frame, executes *forms* in sequence, then restores the earlier window configuration. The return value is the value of the final form in *forms*.

Most Lisp code should not use this macro; `save-selected-window` is typically sufficient. In particular, this macro cannot reliably prevent the code in *forms* from opening new windows, because new windows might be opened in other frames (see Section 17.12 [Choosing Window], page 313), and `save-window-excursion` only saves and restores the window configuration on the current frame.

Do not use this macro in `window-size-change-functions`; exiting the macro triggers execution of `window-size-change-functions`, leading to an endless loop.

`window-configuration-p` *object* [Function]

This function returns `t` if *object* is a window configuration.

`compare-window-configurations` *config1 config2* [Function]

This function compares two window configurations as regards the structure of windows, but ignores the values of point and mark and the saved scrolling positions—it can return `t` even if those aspects differ.

The function `equal` can also compare two window configurations; it regards configurations as unequal if they differ in any respect, even a saved point or mark.

`window-configuration-frame` *config* [Function]

This function returns the frame for which the window configuration *config* was made.

Other primitives to look inside of window configurations would make sense, but are not implemented because we did not need them. See the file '`winner.el`' for some more operations on windows configurations.

The objects returned by `current-window-configuration` die together with the Emacs process. In order to store a window configuration on disk and read it back in another Emacs session, you can use the functions described next. These functions are also useful to clone the state of a frame into an arbitrary live window (`set-window-configuration` effectively clones the windows of a frame into the root window of that very frame only).

`window-state-get` **&optional** *window writable* [Function]
This function returns the state of *window* as a Lisp object. The argument *window* must be a valid window and defaults to the root window of the selected frame.

If the optional argument *writable* is non-`nil`, this means to not use markers for sampling positions like `window-point` or `window-start`. This argument should be non-`nil` when the state will be written to disk and read back in another session.

Together, the argument *writable* and the variable `window-persistent-parameters` specify which window parameters are saved by this function. See Section 17.25 [Window Parameters], page 337.

The value returned by `window-state-get` can be used in the same session to make a clone of a window in another window. It can be also written to disk and read back in another session. In either case, use the following function to restore the state of the window.

`window-state-put` *state* **&optional** *window ignore* [Function]
This function puts the window state *state* into *window*. The argument *state* should be the state of a window returned by an earlier invocation of `window-state-get`, see above. The optional argument *window* must specify a live window and defaults to the selected one.

If the optional argument *ignore* is non-`nil`, it means to ignore minimum window sizes and fixed-size restrictions. If *ignore* is `safe`, this means windows can get as small as one line and/or two columns.

## 17.25 Window Parameters

This section describes how window parameters can be used to associate additional information with windows.

`window-parameter` *window parameter* [Function]
This function returns *window*'s value for *parameter*. The default for *window* is the selected window. If *window* has no setting for *parameter*, this function returns `nil`.

`window-parameters` **&optional** *window* [Function]
This function returns all parameters of *window* and their values. The default for *window* is the selected window. The return value is either `nil`, or an association list whose elements have the form `(`*`parameter`*` . `*`value`*`)`.

`set-window-parameter` *window parameter value* [Function]
This function sets *window*'s value of *parameter* to *value* and returns *value*. The default for *window* is the selected window.

By default, the functions that save and restore window configurations or the states of windows (see Section 17.24 [Window Configurations], page 335) do not care about window parameters. This means that when you change the value of a parameter within the body of a `save-window-excursion`, the previous value is not restored when that macro exits. It also means that when you restore via `window-state-put` a window state saved earlier by `window-state-get`, all cloned windows have their parameters reset to `nil`. The following variable allows you to override the standard behavior:

`window-persistent-parameters` [Variable]

This variable is an alist specifying which parameters get saved by `current-window-configuration` and `window-state-get`, and subsequently restored by `set-window-configuration` and `window-state-put`. See Section 17.24 [Window Configurations], page 335.

The CAR of each entry of this alist is a symbol specifying the parameter. The CDR should be one of the following:

`nil` This value means the parameter is saved neither by `window-state-get` nor by `current-window-configuration`.

`t` This value specifies that the parameter is saved by `current-window-configuration` and (provided its *writable* argument is `nil`) by `window-state-get`.

`writable` This means that the parameter is saved unconditionally by both `current-window-configuration` and `window-state-get`. This value should not be used for parameters whose values do not have a read syntax. Otherwise, invoking `window-state-put` in another session may fail with an `invalid-read-syntax` error.

Some functions (notably `delete-window`, `delete-other-windows` and `split-window`), may behave specially when their *window* argument has a parameter set. You can override such special behavior by binding the following variable to a non-`nil` value:

`ignore-window-parameters` [Variable]

If this variable is non-`nil`, some standard functions do not process window parameters. The functions currently affected by this are `split-window`, `delete-window`, `delete-other-windows`, and `other-window`.

An application can bind this variable to a non-`nil` value around calls to these functions. If it does so, the application is fully responsible for correctly assigning the parameters of all involved windows when exiting that function.

The following parameters are currently used by the window management code:

`delete-window`

This parameter affects the execution of `delete-window` (see Section 17.6 [Deleting Windows], page 300).

`delete-other-windows`

This parameter affects the execution of `delete-other-windows` (see Section 17.6 [Deleting Windows], page 300).

`split-window`

This parameter affects the execution of `split-window` (see Section 17.5 [Splitting Windows], page 297).

`other-window`

This parameter affects the execution of `other-window` (see Section 17.9 [Cyclic Window Ordering], page 307).

`no-other-window`
: This parameter marks the window as not selectable by `other-window` (see Section 17.9 [Cyclic Window Ordering], page 307).

`clone-of`
: This parameter specifies the window that this one has been cloned from. It is installed by `window-state-get` (see Section 17.24 [Window Configurations], page 335).

`quit-restore`
: This parameter is installed by the buffer display functions (see Section 17.12 [Choosing Window], page 313) and consulted by `quit-restore-window` (see Section 17.17 [Quitting Windows], page 321). It contains four elements:

  The first element is one of the symbols `window`, meaning that the window has been specially created by `display-buffer`; `frame`, a separate frame has been created; `same`, the window has displayed the same buffer before; or `other`, the window showed another buffer before.

  The second element is either one of the symbols `window` or `frame`, or a list whose elements are the buffer shown in the window before, that buffer's window start and window point positions, and the window's height at that time.

  The third element is the window selected at the time the parameter was created. The function `quit-restore-window` tries to reselect that window when it deletes the window passed to it as argument.

  The fourth element is the buffer whose display caused the creation of this parameter. `quit-restore-window` deletes the specified window only if it still shows that buffer.

There are additional parameters `window-atom` and `window-side`; these are reserved and should not be used by applications.

## 17.26 Hooks for Window Scrolling and Changes

This section describes how a Lisp program can take action whenever a window displays a different part of its buffer or a different buffer. There are three actions that can change this: scrolling the window, switching buffers in the window, and changing the size of the window. The first two actions run `window-scroll-functions`; the last runs `window-size-change-functions`.

`window-scroll-functions` [Variable]
: This variable holds a list of functions that Emacs should call before redisplaying a window with scrolling. Displaying a different buffer in the window also runs these functions.

  This variable is not a normal hook, because each function is called with two arguments: the window, and its new display-start position.

  These functions must take care when using `window-end` (see Section 17.19 [Window Start and End], page 324); if you need an up-to-date value, you must use the *update* argument to ensure you get it.

  **Warning:** don't use this feature to alter the way the window is scrolled. It's not designed for that, and such use probably won't work.

`window-size-change-functions` [Variable]

This variable holds a list of functions to be called if the size of any window changes for any reason. The functions are called just once per redisplay, and just once for each frame on which size changes have occurred.

Each function receives the frame as its sole argument. There is no direct way to find out which windows on that frame have changed size, or precisely how. However, if a size-change function records, at each call, the existing windows and their sizes, it can also compare the present sizes and the previous sizes.

Creating or deleting windows counts as a size change, and therefore causes these functions to be called. Changing the frame size also counts, because it changes the sizes of the existing windows.

You may use `save-selected-window` in these functions (see Section 17.8 [Selecting Windows], page 306). However, do not use `save-window-excursion` (see Section 17.24 [Window Configurations], page 335); exiting that macro counts as a size change, which would cause these functions to be called over and over.

`window-configuration-change-hook` [Variable]

A normal hook that is run every time you change the window configuration of an existing frame. This includes splitting or deleting windows, changing the sizes of windows, or displaying a different buffer in a window.

The buffer-local part of this hook is run once for each window on the affected frame, with the relevant window selected and its buffer current. The global part is run once for the modified frame, with that frame selected.

In addition, you can use `jit-lock-register` to register a Font Lock fontification function, which will be called whenever parts of a buffer are (re)fontified because a window was scrolled or its size changed. See Section 20.6.4 [Other Font Lock Variables], page 435.

# 18 Frames

A *frame* is a screen object that contains one or more Emacs windows (see Chapter 17 [Windows], page 289). It is the kind of object called a "window" in the terminology of graphical environments; but we can't call it a "window" here, because Emacs uses that word in a different way. In Emacs Lisp, a *frame object* is a Lisp object that represents a frame on the screen. See ⟨undefined⟩ [Frame Type], page ⟨undefined⟩.

A frame initially contains a single main window and/or a minibuffer window; you can subdivide the main window vertically or horizontally into smaller windows. See Section 17.5 [Splitting Windows], page 297.

A *terminal* is a display device capable of displaying one or more Emacs frames. In Emacs Lisp, a *terminal object* is a Lisp object that represents a terminal. See ⟨undefined⟩ [Terminal Type], page ⟨undefined⟩.

There are two classes of terminals: *text terminals* and *graphical terminals*. Text terminals are non-graphics-capable displays, including `xterm` and other terminal emulators. On a text terminal, each Emacs frame occupies the terminal's entire screen; although you can create additional frames and switch between them, the terminal only shows one frame at a time. Graphical terminals, on the other hand, are managed by graphical display systems such as the X Window System, which allow Emacs to show multiple frames simultaneously on the same display.

On GNU and Unix systems, you can create additional frames on any available terminal, within a single Emacs session, regardless of whether Emacs was started on a text or graphical terminal. Emacs can display on both graphical and text terminals simultaneously. This comes in handy, for instance, when you connect to the same session from several remote locations. See Section 18.2 [Multiple Terminals], page 342.

`framep` *object* [Function]

This predicate returns a non-`nil` value if *object* is a frame, and `nil` otherwise. For a frame, the value indicates which kind of display the frame uses:

| | |
|---|---|
| `t` | The frame is displayed on a text terminal. |
| `x` | The frame is displayed on an X graphical terminal. |
| `w32` | The frame is displayed on a MS-Windows graphical terminal. |
| `ns` | The frame is displayed on a GNUstep or Macintosh Cocoa graphical terminal. |
| `pc` | The frame is displayed on an MS-DOS terminal. |

`frame-terminal` **&optional** *frame* [Function]

This function returns the terminal object that displays *frame*. If *frame* is `nil` or unspecified, it defaults to the selected frame.

`terminal-live-p` *object* [Function]

This predicate returns a non-`nil` value if *object* is a terminal that is live (i.e., not deleted), and `nil` otherwise. For live terminals, the return value indicates what kind of frames are displayed on that terminal; the list of possible values is the same as for `framep` above.

## 18.1 Creating Frames

To create a new frame, call the function `make-frame`.

`make-frame` **&optional** *alist* [Command]
This function creates and returns a new frame, displaying the current buffer.

The *alist* argument is an alist that specifies frame parameters for the new frame. See Section 18.3 [Frame Parameters], page 345. If you specify the `terminal` parameter in *alist*, the new frame is created on that terminal. Otherwise, if you specify the `window-system` frame parameter in *alist*, that determines whether the frame should be displayed on a text terminal or a graphical terminal. See Section 11.22 [Window Systems], page 194. If neither is specified, the new frame is created in the same terminal as the selected frame.

Any parameters not mentioned in *alist* default to the values in the alist `default-frame-alist` (see Section 18.3.2 [Initial Parameters], page 345); parameters not specified there default from the X resources or its equivalent on your operating system (see Section "X Resources" in *The GNU Emacs Manual*). After the frame is created, Emacs applies any parameters listed in `frame-inherited-parameters` (see below) and not present in the argument, taking the values from the frame that was selected when `make-frame` was called.

This function itself does not make the new frame the selected frame. See Section 18.9 [Input Focus], page 358. The previously selected frame remains selected. On graphical terminals, however, the windowing system may select the new frame for its own reasons.

`before-make-frame-hook` [Variable]
A normal hook run by `make-frame` before it creates the frame.

`after-make-frame-functions` [Variable]
An abnormal hook run by `make-frame` after it creates the frame. Each function in `after-make-frame-functions` receives one argument, the frame just created.

`frame-inherited-parameters` [Variable]
This variable specifies the list of frame parameters that a newly created frame inherits from the currently selected frame. For each parameter (a symbol) that is an element in the list and is not present in the argument to `make-frame`, the function sets the value of that parameter in the created frame to its value in the selected frame.

## 18.2 Multiple Terminals

Emacs represents each terminal as a *terminal object* data type (see ⟨undefined⟩ [Terminal Type], page ⟨undefined⟩). On GNU and Unix systems, Emacs can use multiple terminals simultaneously in each session. On other systems, it can only use a single terminal. Each terminal object has the following attributes:

- The name of the device used by the terminal (e.g., '`:0.0`' or '`/dev/tty`').
- The terminal and keyboard coding systems used on the terminal. See ⟨undefined⟩ [Terminal I/O Encoding], page ⟨undefined⟩.

- The kind of display associated with the terminal. This is the symbol returned by the function `terminal-live-p` (i.e., `x`, `t`, `w32`, `ns`, or `pc`). See Chapter 18 [Frames], page 341.
- A list of terminal parameters. See Section 18.4 [Terminal Parameters], page 355.

There is no primitive for creating terminal objects. Emacs creates them as needed, such as when you call `make-frame-on-display` (described below).

`terminal-name` **&optional** *terminal* [Function]
: This function returns the file name of the device used by *terminal*. If *terminal* is omitted or `nil`, it defaults to the selected frame's terminal. *terminal* can also be a frame, meaning that frame's terminal.

`terminal-list` [Function]
: This function returns a list of all live terminal objects.

`get-device-terminal` *device* [Function]
: This function returns a terminal whose device name is given by *device*. If *device* is a string, it can be either the file name of a terminal device, or the name of an X display of the form '*host*`:`*server*`.`*screen*'. If *device* is a frame, this function returns that frame's terminal; `nil` means the selected frame. Finally, if *device* is a terminal object that represents a live terminal, that terminal is returned. The function signals an error if its argument is none of the above.

`delete-terminal` **&optional** *terminal force* [Function]
: This function deletes all frames on *terminal* and frees the resources used by it. It runs the abnormal hook `delete-terminal-functions`, passing *terminal* as the argument to each function.

  If *terminal* is omitted or `nil`, it defaults to the selected frame's terminal. *terminal* can also be a frame, meaning that frame's terminal.

  Normally, this function signals an error if you attempt to delete the sole active terminal, but if *force* is non-`nil`, you are allowed to do so. Emacs automatically calls this function when the last frame on a terminal is deleted (see Section 18.6 [Deleting Frames], page 357).

`delete-terminal-functions` [Variable]
: An abnormal hook run by `delete-terminal`. Each function receives one argument, the *terminal* argument passed to `delete-terminal`. Due to technical details, the functions may be called either just before the terminal is deleted, or just afterwards.

A few Lisp variables are *terminal-local*; that is, they have a separate binding for each terminal. The binding in effect at any time is the one for the terminal that the currently selected frame belongs to. These variables include `default-minibuffer-frame`, `defining-kbd-macro`, `last-kbd-macro`, and `system-key-alist`. They are always terminal-local, and can never be buffer-local (see ⟨undefined⟩ [Buffer-Local Variables], page ⟨undefined⟩).

On GNU and Unix systems, each X display is a separate graphical terminal. When Emacs is started from within the X window system, it uses the X display specified by the `DISPLAY` environment variable, or by the '`--display`' option (see Section "Initial Options"

in *The GNU Emacs Manual*). Emacs can connect to other X displays via the command `make-frame-on-display`. Each X display has its own selected frame and its own minibuffer windows; however, only one of those frames is "*the* selected frame" at any given moment (see Section 18.9 [Input Focus], page 358). Emacs can even connect to other text terminals, by interacting with the `emacsclient` program. See Section "Emacs Server" in *The GNU Emacs Manual*.

A single X server can handle more than one display. Each X display has a three-part name, '*host*`:`*server*`.`*screen*'. The first two parts, *host* and *server*, identify the X server; the third part, *screen*, identifies a screen number on that X server. When you use two or more screens belonging to one server, Emacs knows by the similarity in their names that they share a single keyboard.

On some "multi-monitor" setups, a single X display outputs to more than one physical monitor. Currently, there is no way for Emacs to distinguish between the different physical monitors.

`make-frame-on-display` *display* **&optional** *parameters* [Command]
: This function creates and returns a new frame on *display*, taking the other frame parameters from the alist *parameters*. *display* should be the name of an X display (a string).

  Before creating the frame, this function ensures that Emacs is "set up" to display graphics. For instance, if Emacs has not processed X resources (e.g., if it was started on a text terminal), it does so at this time. In all other respects, this function behaves like `make-frame` (see Section 18.1 [Creating Frames], page 342).

`x-display-list` [Function]
: This function returns a list that indicates which X displays Emacs has a connection to. The elements of the list are strings, and each one is a display name.

`x-open-connection` *display* **&optional** *xrm-string must-succeed* [Function]
: This function opens a connection to the X display *display*, without creating a frame on that display. Normally, Emacs Lisp programs need not call this function, as `make-frame-on-display` calls it automatically. The only reason for calling it is to check whether communication can be established with a given X display.

  The optional argument *xrm-string*, if not `nil`, is a string of resource names and values, in the same format used in the '`.Xresources`' file. See Section "X Resources" in *The GNU Emacs Manual*. These values apply to all Emacs frames created on this display, overriding the resource values recorded in the X server. Here's an example of what this string might look like:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

  If *must-succeed* is non-`nil`, failure to open the connection terminates Emacs. Otherwise, it is an ordinary Lisp error.

`x-close-connection` *display* [Function]
: This function closes the connection to display *display*. Before you can do this, you must first delete all the frames that were open on that display (see Section 18.6 [Deleting Frames], page 357).

## 18.3 Frame Parameters

A frame has many parameters that control its appearance and behavior. Just what parameters a frame has depends on what display mechanism it uses.

Frame parameters exist mostly for the sake of graphical displays. Most frame parameters have no effect when applied to a frame on a text terminal; only the `height`, `width`, `name`, `title`, `menu-bar-lines`, `buffer-list` and `buffer-predicate` parameters do something special. If the terminal supports colors, the parameters `foreground-color`, `background-color`, `background-mode` and `display-type` are also meaningful. If the terminal supports frame transparency, the parameter `alpha` is also meaningful.

### 18.3.1 Access to Frame Parameters

These functions let you read and change the parameter values of a frame.

`frame-parameter` *frame parameter* [Function]
: This function returns the value of the parameter *parameter* (a symbol) of *frame*. If *frame* is `nil`, it returns the selected frame's parameter. If *frame* has no setting for *parameter*, this function returns `nil`.

`frame-parameters` **&optional** *frame* [Function]
: The function `frame-parameters` returns an alist listing all the parameters of *frame* and their values. If *frame* is `nil` or omitted, this returns the selected frame's parameters

`modify-frame-parameters` *frame alist* [Function]
: This function alters the parameters of frame *frame* based on the elements of *alist*. Each element of *alist* has the form `(parm . value)`, where *parm* is a symbol naming a parameter. If you don't mention a parameter in *alist*, its value doesn't change. If *frame* is `nil`, it defaults to the selected frame.

`set-frame-parameter` *frame parm value* [Function]
: This function sets the frame parameter *parm* to the specified *value*. If *frame* is `nil`, it defaults to the selected frame.

`modify-all-frames-parameters` *alist* [Function]
: This function alters the frame parameters of all existing frames according to *alist*, then modifies `default-frame-alist` (and, if necessary, `initial-frame-alist`) to apply the same parameter values to frames that will be created henceforth.

### 18.3.2 Initial Frame Parameters

You can specify the parameters for the initial startup frame by setting `initial-frame-alist` in your init file (see Section 33.4 [Init File], page 711).

`initial-frame-alist` [User Option]
: This variable's value is an alist of parameter values used when creating the initial frame. You can set this variable to specify the appearance of the initial frame without altering subsequent frames. Each element has the form:

```
(parameter . value)
```

Emacs creates the initial frame before it reads your init file. After reading that file, Emacs checks `initial-frame-alist`, and applies the parameter settings in the altered value to the already created initial frame.

If these settings affect the frame geometry and appearance, you'll see the frame appear with the wrong ones and then change to the specified ones. If that bothers you, you can specify the same geometry and appearance with X resources; those do take effect before the frame is created. See Section "X Resources" in *The GNU Emacs Manual*.

X resource settings typically apply to all frames. If you want to specify some X resources solely for the sake of the initial frame, and you don't want them to apply to subsequent frames, here's how to achieve this. Specify parameters in `default-frame-alist` to override the X resources for subsequent frames; then, to prevent these from affecting the initial frame, specify the same parameters in `initial-frame-alist` with values that match the X resources.

If these parameters include `(minibuffer . nil)`, that indicates that the initial frame should have no minibuffer. In this case, Emacs creates a separate *minibuffer-only frame* as well.

`minibuffer-frame-alist` [User Option]

This variable's value is an alist of parameter values used when creating an initial minibuffer-only frame (i.e., the minibuffer-only frame that Emacs creates if `initial-frame-alist` specifies a frame with no minibuffer).

`default-frame-alist` [User Option]

This is an alist specifying default values of frame parameters for all Emacs frames—the first frame, and subsequent frames. When using the X Window System, you can get the same results by means of X resources in many cases.

Setting this variable does not affect existing frames. Furthermore, functions that display a buffer in a separate frame may override the default parameters by supplying their own parameters.

If you invoke Emacs with command-line options that specify frame appearance, those options take effect by adding elements to either `initial-frame-alist` or `default-frame-alist`. Options which affect just the initial frame, such as '`-geometry`' and '`--maximized`', add to `initial-frame-alist`; the others add to `default-frame-alist`. see Section "Command Line Arguments for Emacs Invocation" in *The GNU Emacs Manual*.

### 18.3.3 Window Frame Parameters

Just what parameters a frame has depends on what display mechanism it uses. This section describes the parameters that have special meanings on some or all kinds of terminals. Of these, `name`, `title`, `height`, `width`, `buffer-list` and `buffer-predicate` provide meaningful information in terminal frames, and `tty-color-mode` is meaningful only for frames on text terminals.

#### 18.3.3.1 Basic Parameters

These frame parameters give the most basic information about the frame. `title` and `name` are meaningful on all terminals.

`display` The display on which to open this frame. It should be a string of the form `"host:dpy.screen"`, just like the `DISPLAY` environment variable.

`display-type`
This parameter describes the range of possible colors that can be used in this frame. Its value is `color`, `grayscale` or `mono`.

`title` If a frame has a non-`nil` title, it appears in the window system's title bar at the top of the frame, and also in the mode line of windows in that frame if `mode-line-frame-identification` uses '`%F`' (see Section 20.4.5 [%-Constructs], page 424). This is normally the case when Emacs is not using a window system, and can only display one frame at a time. See Section 18.5 [Frame Titles], page 356.

`name` The name of the frame. The frame name serves as a default for the frame title, if the `title` parameter is unspecified or `nil`. If you don't specify a name, Emacs sets the frame name automatically (see Section 18.5 [Frame Titles], page 356).

If you specify the frame name explicitly when you create the frame, the name is also used (instead of the name of the Emacs executable) when looking up X resources for the frame.

`explicit-name`
If the frame name was specified explicitly when the frame was created, this parameter will be that name. If the frame wasn't explicitly named, this parameter will be `nil`.

### 18.3.3.2 Position Parameters

Position parameters' values are normally measured in pixels, but on text terminals they count characters or lines instead.

`left` The position, in pixels, of the left (or right) edge of the frame with respect to the left (or right) edge of the screen. The value may be:

an integer A positive integer relates the left edge of the frame to the left edge of the screen. A negative integer relates the right frame edge to the right screen edge.

`(+ pos)` This specifies the position of the left frame edge relative to the left screen edge. The integer *pos* may be positive or negative; a negative value specifies a position outside the screen.

`(- pos)` This specifies the position of the right frame edge relative to the right screen edge. The integer *pos* may be positive or negative; a negative value specifies a position outside the screen.

Some window managers ignore program-specified positions. If you want to be sure the position you specify is not ignored, specify a non-`nil` value for the `user-position` parameter as well.

`top` The screen position of the top (or bottom) edge, in pixels, with respect to the top (or bottom) edge of the screen. It works just like `left`, except vertically instead of horizontally.

`icon-left`
: The screen position of the left edge of the frame's icon, in pixels, counting from the left edge of the screen. This takes effect when the frame is iconified, if the window manager supports this feature. If you specify a value for this parameter, then you must also specify a value for `icon-top` and vice versa.

`icon-top`
: The screen position of the top edge of the frame's icon, in pixels, counting from the top edge of the screen. This takes effect when the frame is iconified, if the window manager supports this feature.

`user-position`
: When you create a frame and specify its screen position with the `left` and `top` parameters, use this parameter to say whether the specified position was user-specified (explicitly requested in some way by a human user) or merely program-specified (chosen by a program). A non-`nil` value says the position was user-specified.

  Window managers generally heed user-specified positions, and some heed program-specified positions too. But many ignore program-specified positions, placing the window in a default fashion or letting the user place it with the mouse. Some window managers, including `twm`, let the user specify whether to obey program-specified positions or ignore them.

  When you call `make-frame`, you should specify a non-`nil` value for this parameter if the values of the `left` and `top` parameters represent the user's stated preference; otherwise, use `nil`.

#### 18.3.3.3 Size Parameters

Frame parameters specify frame sizes in character units. On graphical displays, the `default` face determines the actual pixel sizes of these character units (see Section 11.12.1 [Face Attributes], page 138).

`height`
: The height of the frame contents, in characters. (To get the height in pixels, call `frame-pixel-height`; see Section 18.3.4 [Size and Position], page 354.)

`width`
: The width of the frame contents, in characters. (To get the width in pixels, call `frame-pixel-width`; see Section 18.3.4 [Size and Position], page 354.)

`user-size`
: This does for the size parameters `height` and `width` what the `user-position` parameter (see Section 18.3.3.2 [Position Parameters], page 347) does for the position parameters `top` and `left`.

`fullscreen`
: Specify that width, height or both shall be maximized. The value `fullwidth` specifies that width shall be as wide as possible. The value `fullheight` specifies that height shall be as tall as possible. The value `fullboth` specifies that both the width and the height shall be set to the size of the screen. The value `maximized` specifies that the frame shall be maximized. The difference between `maximized` and `fullboth` is that the former still has window manager decorations while the latter really covers the whole screen.

#### 18.3.3.4 Layout Parameters

These frame parameters enable or disable various parts of the frame, or control their sizes.

`border-width`
: The width in pixels of the frame's border.

`internal-border-width`
: The distance in pixels between text (or fringe) and the frame's border.

`vertical-scroll-bars`
: Whether the frame has scroll bars for vertical scrolling, and which side of the frame they should be on. The possible values are `left`, `right`, and `nil` for no scroll bars.

`scroll-bar-width`
: The width of vertical scroll bars, in pixels, or `nil` meaning to use the default width.

`left-fringe`
`right-fringe`
: The default width of the left and right fringes of windows in this frame (see Section 11.13 [Fringes], page 156). If either of these is zero, that effectively removes the corresponding fringe.

  When you use `frame-parameter` to query the value of either of these two frame parameters, the return value is always an integer. When using `set-frame-parameter`, passing a `nil` value imposes an actual default value of 8 pixels.

  The combined fringe widths must add up to an integral number of columns, so the actual default fringe widths for the frame, as reported by `frame-parameter`, may be larger than what you specify. Any extra width is distributed evenly between the left and right fringe. However, you can force one fringe or the other to a precise width by specifying that width as a negative integer. If both widths are negative, only the left fringe gets the specified width.

`menu-bar-lines`
: The number of lines to allocate at the top of the frame for a menu bar. The default is 1 if Menu Bar mode is enabled, and 0 otherwise. See Section "Menu Bars" in *The GNU Emacs Manual*.

`tool-bar-lines`
: The number of lines to use for the tool bar. The default is 1 if Tool Bar mode is enabled, and 0 otherwise. See Section "Tool Bars" in *The GNU Emacs Manual*.

`tool-bar-position`
: The position of the tool bar. Currently only for the GTK tool bar. Value can be one of `top`, `bottom` `left`, `right`. The default is `top`.

`line-spacing`
: Additional space to leave below each text line, in pixels (a positive integer). See Section 11.11 [Line Height], page 136, for more information.

#### 18.3.3.5 Buffer Parameters

These frame parameters, meaningful on all kinds of terminals, deal with which buffers have been, or should, be displayed in the frame.

`minibuffer`
: Whether this frame has its own minibuffer. The value `t` means yes, `nil` means no, `only` means this frame is just a minibuffer. If the value is a minibuffer window (in some other frame), the frame uses that minibuffer.

  This frame parameter takes effect when the frame is created, and can not be changed afterwards.

`buffer-predicate`
: The buffer-predicate function for this frame. The function `other-buffer` uses this predicate (from the selected frame) to decide which buffers it should consider, if the predicate is not `nil`. It calls the predicate with one argument, a buffer, once for each buffer; if the predicate returns a non-`nil` value, it considers that buffer.

`buffer-list`
: A list of buffers that have been selected in this frame, ordered most-recently-selected first.

`unsplittable`
: If non-`nil`, this frame's window is never split automatically.

#### 18.3.3.6 Window Management Parameters

The following frame parameters control various aspects of the frame's interaction with the window manager. They have no effect on text terminals.

`visibility`
: The state of visibility of the frame. There are three possibilities: `nil` for invisible, `t` for visible, and `icon` for iconified. See Section 18.10 [Visibility of Frames], page 360.

`auto-raise`
: If non-`nil`, Emacs automatically raises the frame when it is selected. Some window managers do not allow this.

`auto-lower`
: If non-`nil`, Emacs automatically lowers the frame when it is deselected. Some window managers do not allow this.

`icon-type`
: The type of icon to use for this frame. If the value is a string, that specifies a file containing a bitmap to use; `nil` specifies no icon (in which case the window manager decides what to show); any other non-`nil` value specifies the default Emacs icon.

`icon-name`
: The name to use in the icon for this frame, when and if the icon appears. If this is `nil`, the frame's title is used.

`window-id`
: The ID number which the graphical display uses for this frame. Emacs assigns this parameter when the frame is created; changing the parameter has no effect on the actual ID number.

`outer-window-id`
: The ID number of the outermost window-system window in which the frame exists. As with `window-id`, changing this parameter has no actual effect.

`wait-for-wm`
: If non-`nil`, tell Xt to wait for the window manager to confirm geometry changes. Some window managers, including versions of Fvwm2 and KDE, fail to confirm, so Xt hangs. Set this to `nil` to prevent hanging with those window managers.

`sticky`
: If non-`nil`, the frame is visible on all virtual desktops on systems with virtual desktops.

#### 18.3.3.7 Cursor Parameters

This frame parameter controls the way the cursor looks.

`cursor-type`
: How to display the cursor. Legitimate values are:

  `box`
  : Display a filled box. (This is the default.)

  `hollow`
  : Display a hollow box.

  `nil`
  : Don't display a cursor.

  `bar`
  : Display a vertical bar between characters.

  `(bar . width)`
  : Display a vertical bar *width* pixels wide between characters.

  `hbar`
  : Display a horizontal bar.

  `(hbar . height)`
  : Display a horizontal bar *height* pixels high.

The `cursor-type` frame parameter may be overridden by the variables `cursor-type` and `cursor-in-non-selected-windows`:

`cursor-type` [Variable]
: This buffer-local variable controls how the cursor looks in a selected window showing the buffer. If its value is `t`, that means to use the cursor specified by the `cursor-type` frame parameter. Otherwise, the value should be one of the cursor types listed above, and it overrides the `cursor-type` frame parameter.

`cursor-in-non-selected-windows` [User Option]
: This buffer-local variable controls how the cursor looks in a window that is not selected. It supports the same values as the `cursor-type` frame parameter; also, `nil` means don't display a cursor in nonselected windows, and `t` (the default) means use a standard modification of the usual cursor type (solid box becomes hollow box, and bar becomes a narrower bar).

`blink-cursor-alist` [User Option]

This variable specifies how to blink the cursor. Each element has the form `(`*`on-state`* `. `*`off-state`*`)`. Whenever the cursor type equals *on-state* (comparing using `equal`), the corresponding *off-state* specifies what the cursor looks like when it blinks "off". Both *on-state* and *off-state* should be suitable values for the `cursor-type` frame parameter.

There are various defaults for how to blink each type of cursor, if the type is not mentioned as an *on-state* here. Changes in this variable do not take effect immediately, only when you specify the `cursor-type` frame parameter.

### 18.3.3.8 Font and Color Parameters

These frame parameters control the use of fonts and colors.

`font-backend`

A list of symbols, specifying the *font backends* to use for drawing fonts in the frame, in order of priority. On X, there are currently two available font backends: `x` (the X core font driver) and `xft` (the Xft font driver). On Windows, there are currently two available font backends: `gdi` and `uniscribe` (see Section "Windows Fonts" in *The GNU Emacs Manual*). On other systems, there is only one available font backend, so it does not make sense to modify this frame parameter.

`background-mode`

This parameter is either `dark` or `light`, according to whether the background color is a light one or a dark one.

`tty-color-mode`

This parameter overrides the terminal's color support as given by the system's terminal capabilities database in that this parameter's value specifies the color mode to use on a text terminal. The value can be either a symbol or a number. A number specifies the number of colors to use (and, indirectly, what commands to issue to produce each color). For example, `(tty-color-mode . 8)` specifies use of the ANSI escape sequences for 8 standard text colors. A value of -1 turns off color support.

If the parameter's value is a symbol, it specifies a number through the value of `tty-color-mode-alist`, and the associated number is used instead.

`screen-gamma`

If this is a number, Emacs performs "gamma correction" which adjusts the brightness of all colors. The value should be the screen gamma of your display, a floating point number.

Usual PC monitors have a screen gamma of 2.2, so color values in Emacs, and in X windows generally, are calibrated to display properly on a monitor with that gamma value. If you specify 2.2 for `screen-gamma`, that means no correction is needed. Other values request correction, designed to make the corrected colors appear on your screen the way they would have appeared without correction on an ordinary monitor with a gamma value of 2.2.

If your monitor displays colors too light, you should specify a `screen-gamma` value smaller than 2.2. This requests correction that makes colors darker. A screen gamma value of 1.5 may give good results for LCD color displays.

`alpha` This parameter specifies the opacity of the frame, on graphical displays that support variable opacity. It should be an integer between 0 and 100, where 0 means completely transparent and 100 means completely opaque. It can also have a `nil` value, which tells Emacs not to set the frame opacity (leaving it to the window manager).

To prevent the frame from disappearing completely from view, the variable `frame-alpha-lower-limit` defines a lower opacity limit. If the value of the frame parameter is less than the value of this variable, Emacs uses the latter. By default, `frame-alpha-lower-limit` is 20.

The `alpha` frame parameter can also be a cons cell `(‘active’ . ‘inactive’)`, where ‘`active`’ is the opacity of the frame when it is selected, and ‘`inactive`’ is the opacity when it is not selected.

The following frame parameters are semi-obsolete in that they are automatically equivalent to particular face attributes of particular faces (see Section “Standard Faces” in *The Emacs Manual*):

`font` The name of the font for displaying text in the frame. This is a string, either a valid font name for your system or the name of an Emacs fontset (see Section 19.14 [Fontsets], page 389). It is equivalent to the `font` attribute of the `default` face.

`foreground-color`
The color to use for the image of a character. It is equivalent to the `:foreground` attribute of the `default` face.

`background-color`
The color to use for the background of characters. It is equivalent to the `:background` attribute of the `default` face.

`mouse-color`
The color for the mouse pointer. It is equivalent to the `:background` attribute of the `mouse` face.

`cursor-color`
The color for the cursor that shows point. It is equivalent to the `:background` attribute of the `cursor` face.

`border-color`
The color for the border of the frame. It is equivalent to the `:background` attribute of the `border` face.

`scroll-bar-foreground`
If non-`nil`, the color for the foreground of scroll bars. It is equivalent to the `:foreground` attribute of the `scroll-bar` face.

`scroll-bar-background`
If non-`nil`, the color for the background of scroll bars. It is equivalent to the `:background` attribute of the `scroll-bar` face.

### 18.3.4 Frame Size And Position

You can read or change the size and position of a frame using the frame parameters `left`, `top`, `height`, and `width`. Whatever geometry parameters you don't specify are chosen by the window manager in its usual fashion.

Here are some special features for working with sizes and positions. (For the precise meaning of "selected frame" used by these functions, see Section 18.9 [Input Focus], page 358.)

`set-frame-position` *frame left top* [Function]

This function sets the position of the top left corner of *frame* to *left* and *top*. These arguments are measured in pixels, and normally count from the top left corner of the screen.

Negative parameter values position the bottom edge of the window up from the bottom edge of the screen, or the right window edge to the left of the right edge of the screen. It would probably be better if the values were always counted from the left and top, so that negative arguments would position the frame partly off the top or left edge of the screen, but it seems inadvisable to change that now.

`frame-height` **&optional** *frame* [Function]
`frame-width` **&optional** *frame* [Function]

These functions return the height and width of *frame*, measured in lines and columns. If you don't supply *frame*, they use the selected frame.

`frame-pixel-height` **&optional** *frame* [Function]
`frame-pixel-width` **&optional** *frame* [Function]

These functions return the height and width of the main display area of *frame*, measured in pixels. If you don't supply *frame*, they use the selected frame. For a text terminal, the results are in characters rather than pixels.

These values include the internal borders, and windows' scroll bars and fringes (which belong to individual windows, not to the frame itself). The exact value of the heights depends on the window-system and toolkit in use. With GTK+, the height does not include any tool bar or menu bar. With the Motif or Lucid toolkits, it includes the tool bar but not the menu bar. In a graphical version with no toolkit, it includes both the tool bar and menu bar. For a text terminal, the result includes the menu bar.

`frame-char-height` **&optional** *frame* [Function]
`frame-char-width` **&optional** *frame* [Function]

These functions return the height and width of a character in *frame*, measured in pixels. The values depend on the choice of font. If you don't supply *frame*, these functions use the selected frame.

`set-frame-size` *frame cols rows* [Function]

This function sets the size of *frame*, measured in characters; *cols* and *rows* specify the new width and height.

To set the size based on values measured in pixels, use `frame-char-height` and `frame-char-width` to convert them to units of characters.

`set-frame-height` *frame lines* **&optional** *pretend* [Function]
This function resizes *frame* to a height of *lines* lines. The sizes of existing windows in *frame* are altered proportionally to fit.

If *pretend* is non-`nil`, then Emacs displays *lines* lines of output in *frame*, but does not change its value for the actual height of the frame. This is only useful on text terminals. Using a smaller height than the terminal actually implements may be useful to reproduce behavior observed on a smaller screen, or if the terminal malfunctions when using its whole screen. Setting the frame height "for real" does not always work, because knowing the correct actual size may be necessary for correct cursor positioning on text terminals.

`set-frame-width` *frame width* **&optional** *pretend* [Function]
This function sets the width of *frame*, measured in characters. The argument *pretend* has the same meaning as in `set-frame-height`.

`fit-frame-to-buffer` **&optional** *frame max-height min-height* [Command]
This command adjusts the height of *frame* (the default is the selected frame) to fit its contents. The optional arguments *max-height* and *min-height* specify the maximum and minimum new frame heights, respectively.

The default minimum height corresponds to `window-min-height`. The default maximum height is the screen height below the current top position of the frame, minus any margin specified by the option `fit-frame-to-buffer-bottom-margin`.

### 18.3.5 Geometry

Here's how to examine the data in an X-style window geometry specification:

`x-parse-geometry` *geom* [Function]
The function `x-parse-geometry` converts a standard X window geometry string to an alist that you can use as part of the argument to `make-frame`.

The alist describes which parameters were specified in *geom*, and gives the values specified for them. Each element looks like `(`*`parameter`*` . `*`value`*`)`. The possible *parameter* values are `left`, `top`, `width`, and `height`.

For the size parameters, the value must be an integer. The position parameter names `left` and `top` are not totally accurate, because some values indicate the position of the right or bottom edges instead. The *value* possibilities for the position parameters are: an integer, a list `(+ `*`pos`*`)`, or a list `(- `*`pos`*`)`; as previously described (see Section 18.3.3.2 [Position Parameters], page 347).

Here is an example:

```
(x-parse-geometry "35x70+0-0")
     ⇒ ((height . 70) (width . 35)
         (top - 0) (left . 0))
```

## 18.4 Terminal Parameters

Each terminal has a list of associated parameters. These *terminal parameters* are mostly a convenient way of storage for terminal-local variables, but some terminal parameters have a special meaning.

This section describes functions to read and change the parameter values of a terminal. They all accept as their argument either a terminal or a frame; the latter means use that frame's terminal. An argument of `nil` means the selected frame's terminal.

`terminal-parameters` **&optional** *terminal* [Function]
This function returns an alist listing all the parameters of *terminal* and their values.

`terminal-parameter` *terminal parameter* [Function]
This function returns the value of the parameter *parameter* (a symbol) of *terminal*. If *terminal* has no setting for *parameter*, this function returns `nil`.

`set-terminal-parameter` *terminal parameter value* [Function]
This function sets the parameter *parm* of *terminal* to the specified *value*, and returns the previous value of that parameter.

Here's a list of a few terminal parameters that have a special meaning:

`background-mode`
The classification of the terminal's background color, either `light` or `dark`.

`normal-erase-is-backspace`
Value is either 1 or 0, depending on whether `normal-erase-is-backspace-mode` is turned on or off on this terminal. See Section "DEL Does Not Delete" in *The Emacs Manual*.

`terminal-initted`
After the terminal is initialized, this is set to the terminal-specific initialization function.

## 18.5 Frame Titles

Every frame has a `name` parameter; this serves as the default for the frame title which window systems typically display at the top of the frame. You can specify a name explicitly by setting the `name` frame property.

Normally you don't specify the name explicitly, and Emacs computes the frame name automatically based on a template stored in the variable `frame-title-format`. Emacs recomputes the name each time the frame is redisplayed.

`frame-title-format` [Variable]
This variable specifies how to compute a name for a frame when you have not explicitly specified one. The variable's value is actually a mode line construct, just like `mode-line-format`, except that the '`%c`' and '`%l`' constructs are ignored. See Section 20.4.2 [Mode Line Data], page 419.

`icon-title-format` [Variable]
This variable specifies how to compute the name for an iconified frame, when you have not explicitly specified the frame title. This title appears in the icon itself.

`multiple-frames` [Variable]
This variable is set automatically by Emacs. Its value is `t` when there are two or more frames (not counting minibuffer-only frames or invisible frames). The default

value of `frame-title-format` uses `multiple-frames` so as to put the buffer name in the frame title only when there is more than one frame.

The value of this variable is not guaranteed to be accurate except while processing `frame-title-format` or `icon-title-format`.

## 18.6 Deleting Frames

A *live frame* is one that has not been deleted. When a frame is deleted, it is removed from its terminal display, although it may continue to exist as a Lisp object until there are no more references to it.

`delete-frame` **&optional** *frame force* [Command]

This function deletes the frame *frame*. Unless *frame* is a tooltip, it first runs the hook `delete-frame-functions` (each function gets one argument, *frame*). By default, *frame* is the selected frame.

A frame cannot be deleted if its minibuffer is used by other frames. Normally, you cannot delete a frame if all other frames are invisible, but if *force* is non-`nil`, then you are allowed to do so.

`frame-live-p` *frame* [Function]

The function `frame-live-p` returns non-`nil` if the frame *frame* has not been deleted. The possible non-`nil` return values are like those of `framep`. See Chapter 18 [Frames], page 341.

Some window managers provide a command to delete a window. These work by sending a special message to the program that operates the window. When Emacs gets one of these commands, it generates a `delete-frame` event, whose normal definition is a command that calls the function `delete-frame`. See Section 2.7.10 [Misc Events], page 31.

## 18.7 Finding All Frames

`frame-list` [Function]

This function returns a list of all the live frames, i.e., those that have not been deleted. It is analogous to `buffer-list` for buffers, and includes frames on all terminals. The list that you get is newly created, so modifying the list doesn't have any effect on the internals of Emacs.

`visible-frame-list` [Function]

This function returns a list of just the currently visible frames. See Section 18.10 [Visibility of Frames], page 360. Frames on text terminals always count as "visible", even though only the selected one is actually displayed.

`next-frame` **&optional** *frame minibuf* [Function]

This function lets you cycle conveniently through all the frames on the current display from an arbitrary starting point. It returns the "next" frame after *frame* in the cycle. If *frame* is omitted or `nil`, it defaults to the selected frame (see Section 18.9 [Input Focus], page 358).

The second argument, *minibuf*, says which frames to consider:

`nil` Exclude minibuffer-only frames.

`visible` Consider all visible frames.

0 Consider all visible or iconified frames.

a window Consider only the frames using that particular window as their minibuffer.

anything else
Consider all frames.

`previous-frame` **&optional** *frame minibuf* [Function]
Like `next-frame`, but cycles through all frames in the opposite direction.

See also `next-window` and `previous-window`, in Section 17.9 [Cyclic Window Ordering], page 307.

## 18.8 Minibuffers and Frames

Normally, each frame has its own minibuffer window at the bottom, which is used whenever that frame is selected. If the frame has a minibuffer, you can get it with `minibuffer-window` (see ⟨undefined⟩ [Definition of minibuffer-window], page ⟨undefined⟩).

However, you can also create a frame with no minibuffer. Such a frame must use the minibuffer window of some other frame. When you create the frame, you can explicitly specify the minibuffer window to use (in some other frame). If you don't, then the minibuffer is found in the frame which is the value of the variable `default-minibuffer-frame`. Its value should be a frame that does have a minibuffer.

If you use a minibuffer-only frame, you might want that frame to raise when you enter the minibuffer. If so, set the variable `minibuffer-auto-raise` to `t`. See Section 18.11 [Raising and Lowering], page 361.

`default-minibuffer-frame` [Variable]
This variable specifies the frame to use for the minibuffer window, by default. It does not affect existing frames. It is always local to the current terminal and cannot be buffer-local. See Section 18.2 [Multiple Terminals], page 342.

## 18.9 Input Focus

At any time, one frame in Emacs is the *selected frame*. The selected window always resides on the selected frame.

When Emacs displays its frames on several terminals (see Section 18.2 [Multiple Terminals], page 342), each terminal has its own selected frame. But only one of these is "*the* selected frame": it's the frame that belongs to the terminal from which the most recent input came. That is, when Emacs runs a command that came from a certain terminal, the selected frame is the one of that terminal. Since Emacs runs only a single command at any given time, it needs to consider only one selected frame at a time; this frame is what we call *the selected frame* in this manual. The display on which the selected frame is shown is the *selected frame's display*.

`selected-frame` [Function]
This function returns the selected frame.

Some window systems and window managers direct keyboard input to the window object that the mouse is in; others require explicit clicks or commands to *shift the focus* to various window objects. Either way, Emacs automatically keeps track of which frame has the focus. To explicitly switch to a different frame from a Lisp function, call `select-frame-set-input-focus`.

Lisp programs can also switch frames "temporarily" by calling the function `select-frame`. This does not alter the window system's concept of focus; rather, it escapes from the window manager's control until that control is somehow reasserted.

When using a text terminal, only one frame can be displayed at a time on the terminal, so after a call to `select-frame`, the next redisplay actually displays the newly selected frame. This frame remains selected until a subsequent call to `select-frame`. Each frame on a text terminal has a number which appears in the mode line before the buffer name (see Section 20.4.4 [Mode Line Variables], page 422).

`select-frame-set-input-focus` *frame* **&optional** *norecord* [Function]
: This function selects *frame*, raises it (should it happen to be obscured by other frames) and tries to give it the X server's focus. On a text terminal, the next redisplay displays the new frame on the entire terminal screen. The optional argument *norecord* has the same meaning as for `select-frame` (see below). The return value of this function is not significant.

`select-frame` *frame* **&optional** *norecord* [Command]
: This function selects frame *frame*, temporarily disregarding the focus of the X server if any. The selection of *frame* lasts until the next time the user does something to select a different frame, or until the next time this function is called. (If you are using a window system, the previously selected frame may be restored as the selected frame after return to the command loop, because it still may have the window system's input focus.)

  The specified *frame* becomes the selected frame, and its terminal becomes the selected terminal. This function then calls `select-window` as a subroutine, passing the window selected within *frame* as its first argument and *norecord* as its second argument (hence, if *norecord* is non-`nil`, this avoids changing the order of recently selected windows nor the buffer list). See Section 17.8 [Selecting Windows], page 306.

  This function returns *frame*, or `nil` if *frame* has been deleted.

  In general, you should never use `select-frame` in a way that could switch to a different terminal without switching back when you're done.

Emacs cooperates with the window system by arranging to select frames as the server and window manager request. It does so by generating a special kind of input event, called a *focus* event, when appropriate. The command loop handles a focus event by calling `handle-switch-frame`. See Section 2.7.9 [Focus Events], page 30.

`handle-switch-frame` *frame* [Command]
: This function handles a focus event by selecting frame *frame*.

  Focus events normally do their job by invoking this command. Don't call it for any other reason.

`redirect-frame-focus` *frame* **&optional** *focus-frame* [Function]
This function redirects focus from *frame* to *focus-frame*. This means that *focus-frame* will receive subsequent keystrokes and events intended for *frame*. After such an event, the value of `last-event-frame` will be *focus-frame*. Also, switch-frame events specifying *frame* will instead select *focus-frame*.

If *focus-frame* is omitted or `nil`, that cancels any existing redirection for *frame*, which therefore once again receives its own events.

One use of focus redirection is for frames that don't have minibuffers. These frames use minibuffers on other frames. Activating a minibuffer on another frame redirects focus to that frame. This puts the focus on the minibuffer's frame, where it belongs, even though the mouse remains in the frame that activated the minibuffer.

Selecting a frame can also change focus redirections. Selecting frame `bar`, when `foo` had been selected, changes any redirections pointing to `foo` so that they point to `bar` instead. This allows focus redirection to work properly when the user switches from one frame to another using `select-window`.

This means that a frame whose focus is redirected to itself is treated differently from a frame whose focus is not redirected. `select-frame` affects the former but not the latter.

The redirection lasts until `redirect-frame-focus` is called to change it.

`focus-follows-mouse` [User Option]
This option is how you inform Emacs whether the window manager transfers focus when the user moves the mouse. Non-`nil` says that it does. When this is so, the command `other-frame` moves the mouse to a position consistent with the new selected frame.

## 18.10 Visibility of Frames

A frame on a graphical display may be *visible*, *invisible*, or *iconified*. If it is visible, its contents are displayed in the usual manner. If it is iconified, its contents are not displayed, but there is a little icon somewhere to bring the frame back into view (some window managers refer to this state as *minimized* rather than *iconified*, but from Emacs' point of view they are the same thing). If a frame is invisible, it is not displayed at all.

Visibility is meaningless on text terminals, since only the selected one is actually displayed in any case.

`frame-visible-p` *frame* [Function]
This function returns the visibility status of frame *frame*. The value is `t` if *frame* is visible, `nil` if it is invisible, and `icon` if it is iconified.

On a text terminal, all frames are considered "visible" for the purposes of this function, even though only one frame is displayed. See Section 18.11 [Raising and Lowering], page 361.

`iconify-frame` **&optional** *frame* [Command]
This function iconifies frame *frame*. If you omit *frame*, it iconifies the selected frame.

`make-frame-visible` **&optional** *frame* [Command]
: This function makes frame *frame* visible. If you omit *frame*, it makes the selected frame visible. This does not raise the frame, but you can do that with `raise-frame` if you wish (see Section 18.11 [Raising and Lowering], page 361).

`make-frame-invisible` **&optional** *frame force* [Command]
: This function makes frame *frame* invisible. If you omit *frame*, it makes the selected frame invisible.

  Unless *force* is non-`nil`, this function refuses to make *frame* invisible if all other frames are invisible..

The visibility status of a frame is also available as a frame parameter. You can read or change it as such. See Section 18.3.3.6 [Management Parameters], page 350. The user can also iconify and deiconify frames with the window manager. This happens below the level at which Emacs can exert any control, but Emacs does provide events that you can use to keep track of such changes. See Section 2.7.10 [Misc Events], page 31.

## 18.11 Raising and Lowering Frames

Most window systems use a desktop metaphor. Part of this metaphor is the idea that system-level windows (e.g., Emacs frames) are stacked in a notional third dimension perpendicular to the screen surface. Where two overlap, the one higher up covers the one underneath. You can *raise* or *lower* a frame using the functions `raise-frame` and `lower-frame`.

`raise-frame` **&optional** *frame* [Command]
: This function raises frame *frame* (default, the selected frame). If *frame* is invisible or iconified, this makes it visible.

`lower-frame` **&optional** *frame* [Command]
: This function lowers frame *frame* (default, the selected frame).

`minibuffer-auto-raise` [User Option]
: If this is non-`nil`, activation of the minibuffer raises the frame that the minibuffer window is in.

On window systems, you can also enable auto-raising (on frame selection) or auto-lowering (on frame deselection) using frame parameters. See Section 18.3.3.6 [Management Parameters], page 350.

The concept of raising and lowering frames also applies to text terminal frames. On each text terminal, only the top frame is displayed at any one time.

`tty-top-frame` *terminal* [Function]
: This function returns the top frame on *terminal*. *terminal* should be a terminal object, a frame (meaning that frame's terminal), or `nil` (meaning the selected frame's terminal). If it does not refer to a text terminal, the return value is `nil`.

## 18.12 Frame Configurations

A *frame configuration* records the current arrangement of frames, all their properties, and the window configuration of each one. (See Section 17.24 [Window Configurations], page 335.)

`current-frame-configuration` [Function]
: This function returns a frame configuration list that describes the current arrangement of frames and their contents.

`set-frame-configuration` *configuration* **&optional** *nodelete* [Function]
: This function restores the state of frames described in *configuration*. However, this function does not restore deleted frames.

  Ordinarily, this function deletes all existing frames not listed in *configuration*. But if *nodelete* is non-`nil`, the unwanted frames are iconified instead.

## 18.13 Mouse Tracking

Sometimes it is useful to *track* the mouse, which means to display something to indicate where the mouse is and move the indicator as the mouse moves. For efficient mouse tracking, you need a way to wait until the mouse actually moves.

The convenient way to track the mouse is to ask for events to represent mouse motion. Then you can wait for motion by waiting for an event. In addition, you can easily handle any other sorts of events that may occur. That is useful, because normally you don't want to track the mouse forever—only until some other event, such as the release of a button.

`track-mouse` *body...* [Special Form]
: This special form executes *body*, with generation of mouse motion events enabled. Typically, *body* would use `read-event` to read the motion events and modify the display accordingly. See Section 2.7.8 [Motion Events], page 30, for the format of mouse motion events.

  The value of `track-mouse` is that of the last form in *body*. You should design *body* to return when it sees the up-event that indicates the release of the button, or whatever kind of event means it is time to stop tracking.

The usual purpose of tracking mouse motion is to indicate on the screen the consequences of pushing or releasing a button at the current position.

In many cases, you can avoid the need to track the mouse by using the `mouse-face` text property (see Section 22.19.4 [Special Properties], page 494). That works at a much lower level and runs more smoothly than Lisp-level mouse tracking.

## 18.14 Mouse Position

The functions `mouse-position` and `set-mouse-position` give access to the current position of the mouse.

`mouse-position` [Function]
: This function returns a description of the position of the mouse. The value looks like `(`*`frame x . y`*`)`, where *x* and *y* are integers giving the position in characters relative to the top left corner of the inside of *frame*.

`mouse-position-function` [Variable]
: If non-`nil`, the value of this variable is a function for `mouse-position` to call. `mouse-position` calls this function just before returning, with its normal return value as the sole argument, and it returns whatever this function returns to it.

  This abnormal hook exists for the benefit of packages like '`xt-mouse.el`' that need to do mouse handling at the Lisp level.

`set-mouse-position` *frame x y* [Function]
: This function *warps the mouse* to position *x*, *y* in frame *frame*. The arguments *x* and *y* are integers, giving the position in characters relative to the top left corner of the inside of *frame*. If *frame* is not visible, this function does nothing. The return value is not significant.

`mouse-pixel-position` [Function]
: This function is like `mouse-position` except that it returns coordinates in units of pixels rather than units of characters.

`set-mouse-pixel-position` *frame x y* [Function]
: This function warps the mouse like `set-mouse-position` except that *x* and *y* are in units of pixels rather than units of characters. These coordinates are not required to be within the frame.

  If *frame* is not visible, this function does nothing. The return value is not significant.

`frame-pointer-visible-p` **&optional** *frame* [Function]
: This predicate function returns non-`nil` if the mouse pointer displayed on *frame* is visible; otherwise it returns `nil`. *frame* omitted or `nil` means the selected frame. This is useful when `make-pointer-invisible` is set to `t`: it allows to know if the pointer has been hidden. See Section "Mouse Avoidance" in *The Emacs Manual*.

## 18.15 Pop-Up Menus

When using a window system, a Lisp program can pop up a menu so that the user can choose an alternative with the mouse.

`x-popup-menu` *position menu* [Function]
: This function displays a pop-up menu and returns an indication of what selection the user makes.

  The argument *position* specifies where on the screen to put the top left corner of the menu. It can be either a mouse button event (which says to put the menu where the user actuated the button) or a list of this form:

  ```
  ((xoffset yoffset) window)
  ```

  where *xoffset* and *yoffset* are coordinates, measured in pixels, counting from the top left corner of *window*. *window* may be a window or a frame.

  If *position* is `t`, it means to use the current mouse position. If *position* is `nil`, it means to precompute the key binding equivalents for the keymaps specified in *menu*, without actually displaying or popping up the menu.

The argument *menu* says what to display in the menu. It can be a keymap or a list of keymaps (see ⟨undefined⟩ [Menu Keymaps], page ⟨undefined⟩). In this case, the return value is the list of events corresponding to the user's choice. This list has more than one element if the choice occurred in a submenu. (Note that `x-popup-menu` does not actually execute the command bound to that sequence of events.) On toolkits that support menu titles, the title is taken from the prompt string of *menu* if *menu* is a keymap, or from the prompt string of the first keymap in *menu* if it is a list of keymaps (see ⟨undefined⟩ [Defining Menus], page ⟨undefined⟩).

Alternatively, *menu* can have the following form:

```
(title pane1 pane2...)
```

where each pane is a list of form

```
(title item1 item2...)
```

Each *item* should be a cons cell, `(line . value)`, where *line* is a string and *value* is the value to return if that *line* is chosen. Unlike in a menu keymap, a `nil` *value* does not make the menu item non-selectable. Alternatively, each *item* can be a string rather than a cons cell; this makes a non-selectable menu item.

If the user gets rid of the menu without making a valid choice, for instance by clicking the mouse away from a valid choice or by typing keyboard input, then this normally results in a quit and `x-popup-menu` does not return. But if *position* is a mouse button event (indicating that the user invoked the menu with the mouse) then no quit occurs and `x-popup-menu` returns `nil`.

**Usage note:** Don't use `x-popup-menu` to display a menu if you could do the job with a prefix key defined with a menu keymap. If you use a menu keymap to implement a menu, `C-h c` and `C-h a` can see the individual items in that menu and provide help for them. If instead you implement the menu by defining a command that calls `x-popup-menu`, the help facilities cannot know what happens inside that command, so they cannot give any help for the menu's items.

The menu bar mechanism, which lets you switch between submenus by moving the mouse, cannot look within the definition of a command to see that it calls `x-popup-menu`. Therefore, if you try to implement a submenu using `x-popup-menu`, it cannot work with the menu bar in an integrated fashion. This is why all menu bar submenus are implemented with menu keymaps within the parent menu, and never with `x-popup-menu`. See Section 1.4 [Menu Bar], page 9.

If you want a menu bar submenu to have contents that vary, you should still use a menu keymap to implement it. To make the contents vary, add a hook function to `menu-bar-update-hook` to update the contents of the menu keymap as necessary.

## 18.16 Dialog Boxes

A dialog box is a variant of a pop-up menu—it looks a little different, it always appears in the center of a frame, and it has just one level and one or more buttons. The main use of dialog boxes is for asking questions that the user can answer with "yes", "no", and a few other alternatives. With a single button, they can also force the user to acknowledge important information. The functions `y-or-n-p` and `yes-or-no-p` use dialog boxes instead of the keyboard, when called from commands invoked by mouse clicks.

`x-popup-dialog` *position contents* **&optional** *header* [Function]

This function displays a pop-up dialog box and returns an indication of what selection the user makes. The argument *contents* specifies the alternatives to offer; it has this format:

```
(title (string . value)...)
```

which looks like the list that specifies a single pane for `x-popup-menu`.

The return value is *value* from the chosen alternative.

As for `x-popup-menu`, an element of the list may be just a string instead of a cons cell `(string . value)`. That makes a box that cannot be selected.

If `nil` appears in the list, it separates the left-hand items from the right-hand items; items that precede the `nil` appear on the left, and items that follow the `nil` appear on the right. If you don't include a `nil` in the list, then approximately half the items appear on each side.

Dialog boxes always appear in the center of a frame; the argument *position* specifies which frame. The possible values are as in `x-popup-menu`, but the precise coordinates or the individual window don't matter; only the frame matters.

If *header* is non-`nil`, the frame title for the box is '`Information`', otherwise it is '`Question`'. The former is used for `message-box` (see [message-box], page 115).

In some configurations, Emacs cannot display a real dialog box; so instead it displays the same items in a pop-up menu in the center of the frame.

If the user gets rid of the dialog box without making a valid choice, for instance using the window manager, then this produces a quit and `x-popup-dialog` does not return.

## 18.17 Pointer Shape

You can specify the mouse pointer style for particular text or images using the `pointer` text property, and for images with the `:pointer` and `:map` image properties. The values you can use in these properties are `text` (or `nil`), `arrow`, `hand`, `vdrag`, `hdrag`, `modeline`, and `hourglass`. `text` stands for the usual mouse pointer style used over text.

Over void parts of the window (parts that do not correspond to any of the buffer contents), the mouse pointer usually uses the `arrow` style, but you can specify a different style (one of those above) by setting `void-text-area-pointer`.

`void-text-area-pointer` [User Option]

This variable specifies the mouse pointer style for void text areas. These include the areas after the end of a line or below the last line in the buffer. The default is to use the `arrow` (non-text) pointer style.

When using X, you can specify what the `text` pointer style really looks like by setting the variable `x-pointer-shape`.

`x-pointer-shape` [Variable]

This variable specifies the pointer shape to use ordinarily in the Emacs frame, for the `text` pointer style.

`x-sensitive-text-pointer-shape` [Variable]
: This variable specifies the pointer shape to use when the mouse is over mouse-sensitive text.

These variables affect newly created frames. They do not normally affect existing frames; however, if you set the mouse color of a frame, that also installs the current value of those two variables. See Section 18.3.3.8 [Font and Color Parameters], page 352.

The values you can use, to specify either of these pointer shapes, are defined in the file '`lisp/term/x-win.el`'. Use `M-x apropos RET x-pointer RET` to see a list of them.

## 18.18 Window System Selections

In the X window system, data can be transferred between different applications by means of *selections*. X defines an arbitrary number of *selection types*, each of which can store its own data; however, only three are commonly used: the *clipboard*, *primary selection*, and *secondary selection*. See Section "Cut and Paste" in *The GNU Emacs Manual*, for Emacs commands that make use of these selections. This section documents the low-level functions for reading and setting X selections.

`x-set-selection` *type data* [Command]
: This function sets an X selection. It takes two arguments: a selection type *type*, and the value to assign to it, *data*.

  *type* should be a symbol; it is usually one of `PRIMARY`, `SECONDARY` or `CLIPBOARD`. These are symbols with upper-case names, in accord with X Window System conventions. If *type* is `nil`, that stands for `PRIMARY`.

  If *data* is `nil`, it means to clear out the selection. Otherwise, *data* may be a string, a symbol, an integer (or a cons of two integers or list of two integers), an overlay, or a cons of two markers pointing to the same buffer. An overlay or a pair of markers stands for text in the overlay or between the markers. The argument *data* may also be a vector of valid non-vector selection values.

  This function returns *data*.

`x-get-selection` **&optional** *type data-type* [Function]
: This function accesses selections set up by Emacs or by other X clients. It takes two optional arguments, *type* and *data-type*. The default for *type*, the selection type, is `PRIMARY`.

  The *data-type* argument specifies the form of data conversion to use, to convert the raw data obtained from another X client into Lisp data. Meaningful values include `TEXT`, `STRING`, `UTF8_STRING`, `TARGETS`, `LENGTH`, `DELETE`, `FILE_NAME`, `CHARACTER_POSITION`, `NAME`, `LINE_NUMBER`, `COLUMN_NUMBER`, `OWNER_OS`, `HOST_NAME`, `USER`, `CLASS`, `ATOM`, and `INTEGER`. (These are symbols with upper-case names in accord with X conventions.) The default for *data-type* is `STRING`.

`selection-coding-system` [User Option]
: This variable specifies the coding system to use when reading and writing selections or the clipboard. See Section 19.6 [Coding Systems], page 381. The default is `compound-text-with-extensions`, which converts to the text representation that X11 normally uses.

When Emacs runs on MS-Windows, it does not implement X selections in general, but it does support the clipboard. `x-get-selection` and `x-set-selection` on MS-Windows support the text data type only; if the clipboard holds other types of data, Emacs treats the clipboard as empty.

## 18.19 Drag and Drop

When a user drags something from another application over Emacs, that other application expects Emacs to tell it if Emacs can handle the data that is dragged. The variable `x-dnd-test-function` is used by Emacs to determine what to reply. The default value is `x-dnd-default-test-function` which accepts drops if the type of the data to be dropped is present in `x-dnd-known-types`. You can customize `x-dnd-test-function` and/or `x-dnd-known-types` if you want Emacs to accept or reject drops based on some other criteria.

If you want to change the way Emacs handles drop of different types or add a new type, customize `x-dnd-types-alist`. This requires detailed knowledge of what types other applications use for drag and drop.

When an URL is dropped on Emacs it may be a file, but it may also be another URL type (ftp, http, etc.). Emacs first checks `dnd-protocol-alist` to determine what to do with the URL. If there is no match there and if `browse-url-browser-function` is an alist, Emacs looks for a match there. If no match is found the text for the URL is inserted. If you want to alter Emacs behavior, you can customize these variables.

## 18.20 Color Names

A color name is text (usually in a string) that specifies a color. Symbolic names such as '`black`', '`white`', '`red`', etc., are allowed; use `M-x list-colors-display` to see a list of defined names. You can also specify colors numerically in forms such as '`#rgb`' and '`RGB:r/g/b`', where *r* specifies the red level, *g* specifies the green level, and *b* specifies the blue level. You can use either one, two, three, or four hex digits for *r*; then you must use the same number of hex digits for all *g* and *b* as well, making either 3, 6, 9 or 12 hex digits in all. (See the documentation of the X Window System for more details about numerical RGB specification of colors.)

These functions provide a way to determine which color names are valid, and what they look like. In some cases, the value depends on the *selected frame*, as described below; see Section 18.9 [Input Focus], page 358, for the meaning of the term "selected frame".

To read user input of color names with completion, use `read-color` (see ⟨undefined⟩ [High-Level Completion], page ⟨undefined⟩).

`color-defined-p` *color* **&optional** *frame* [Function]

This function reports whether a color name is meaningful. It returns `t` if so; otherwise, `nil`. The argument *frame* says which frame's display to ask about; if *frame* is omitted or `nil`, the selected frame is used.

Note that this does not tell you whether the display you are using really supports that color. When using X, you can ask for any defined color on any kind of display, and you will get some result—typically, the closest it can do. To determine whether a frame can really display a certain color, use `color-supported-p` (see below).

This function used to be called `x-color-defined-p`, and that name is still supported as an alias.

`defined-colors` **&optional** *frame* [Function]
This function returns a list of the color names that are defined and supported on frame *frame* (default, the selected frame). If *frame* does not support colors, the value is `nil`.

This function used to be called `x-defined-colors`, and that name is still supported as an alias.

`color-supported-p` *color* **&optional** *frame background-p* [Function]
This returns `t` if *frame* can really display the color *color* (or at least something close to it). If *frame* is omitted or `nil`, the question applies to the selected frame.

Some terminals support a different set of colors for foreground and background. If *background-p* is non-`nil`, that means you are asking whether *color* can be used as a background; otherwise you are asking whether it can be used as a foreground.

The argument *color* must be a valid color name.

`color-gray-p` *color* **&optional** *frame* [Function]
This returns `t` if *color* is a shade of gray, as defined on *frame*'s display. If *frame* is omitted or `nil`, the question applies to the selected frame. If *color* is not a valid color name, this function returns `nil`.

`color-values` *color* **&optional** *frame* [Function]
This function returns a value that describes what *color* should ideally look like on *frame*. If *color* is defined, the value is a list of three integers, which give the amount of red, the amount of green, and the amount of blue. Each integer ranges in principle from 0 to 65535, but some displays may not use the full range. This three-element list is called the *rgb values* of the color.

If *color* is not defined, the value is `nil`.

```
(color-values "black")
     ⇒ (0 0 0)
(color-values "white")
     ⇒ (65280 65280 65280)
(color-values "red")
     ⇒ (65280 0 0)
(color-values "pink")
     ⇒ (65280 49152 51968)
(color-values "hungry")
     ⇒ nil
```

The color values are returned for *frame*'s display. If *frame* is omitted or `nil`, the information is returned for the selected frame's display. If the frame cannot display colors, the value is `nil`.

This function used to be called `x-color-values`, and that name is still supported as an alias.

## 18.21 Text Terminal Colors

Text terminals usually support only a small number of colors, and the computer uses small integers to select colors on the terminal. This means that the computer cannot reliably tell what the selected color looks like; instead, you have to inform your application which small integers correspond to which colors. However, Emacs does know the standard set of colors and will try to use them automatically.

The functions described in this section control how terminal colors are used by Emacs.

Several of these functions use or return *rgb values*, described in Section 18.20 [Color Names], page 367.

These functions accept a display (either a frame or the name of a terminal) as an optional argument. We hope in the future to make Emacs support different colors on different text terminals; then this argument will specify which terminal to operate on (the default being the selected frame's terminal; see Section 18.9 [Input Focus], page 358). At present, though, the *frame* argument has no effect.

`tty-color-define` *name number* **&optional** *rgb frame* [Function]
: This function associates the color name *name* with color number *number* on the terminal.

  The optional argument *rgb*, if specified, is an rgb value, a list of three numbers that specify what the color actually looks like. If you do not specify *rgb*, then this color cannot be used by `tty-color-approximate` to approximate other colors, because Emacs will not know what it looks like.

`tty-color-clear` **&optional** *frame* [Function]
: This function clears the table of defined colors for a text terminal.

`tty-color-alist` **&optional** *frame* [Function]
: This function returns an alist recording the known colors supported by a text terminal.

  Each element has the form `(name number . rgb)` or `(name number)`. Here, *name* is the color name, *number* is the number used to specify it to the terminal. If present, *rgb* is a list of three color values (for red, green, and blue) that says what the color actually looks like.

`tty-color-approximate` *rgb* **&optional** *frame* [Function]
: This function finds the closest color, among the known colors supported for *display*, to that described by the rgb value *rgb* (a list of color values). The return value is an element of `tty-color-alist`.

`tty-color-translate` *color* **&optional** *frame* [Function]
: This function finds the closest color to *color* among the known colors supported for *display* and returns its index (an integer). If the name *color* is not defined, the value is `nil`.

## 18.22 X Resources

This section describes some of the functions and variables for querying and using X resources, or their equivalent on your operating system. See Section "X Resources" in *The GNU Emacs Manual*, for more information about X resources.

`x-get-resource` *attribute class* **&optional** *component subclass* [Function]
: The function `x-get-resource` retrieves a resource value from the X Window defaults database.

  Resources are indexed by a combination of a *key* and a *class*. This function searches using a key of the form '*`instance.attribute`*' (where *instance* is the name under which Emacs was invoked), and using '`Emacs.`*`class`*' as the class.

  The optional arguments *component* and *subclass* add to the key and the class, respectively. You must specify both of them or neither. If you specify them, the key is '*`instance.component.attribute`*', and the class is '`Emacs.`*`class.subclass`*'.

`x-resource-class` [Variable]
: This variable specifies the application name that `x-get-resource` should look up. The default value is `"Emacs"`. You can examine X resources for application names other than "Emacs" by binding this variable to some other string, around a call to `x-get-resource`.

`x-resource-name` [Variable]
: This variable specifies the instance name that `x-get-resource` should look up. The default value is the name Emacs was invoked with, or the value specified with the '`-name`' or '`-rn`' switches.

To illustrate some of the above, suppose that you have the line:

```
xterm.vt100.background: yellow
```

in your X resources file (whose name is usually '`~/.Xdefaults`' or '`~/.Xresources`'). Then:

```
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "vt100.background" "VT100.Background"))
     ⇒ "yellow"
(let ((x-resource-class "XTerm") (x-resource-name "xterm"))
  (x-get-resource "background" "VT100" "vt100" "Background"))
     ⇒ "yellow"
```

`inhibit-x-resources` [Variable]
: If this variable is non-`nil`, Emacs does not look up X resources, and X resources do not have any effect when creating new frames.

## 18.23 Display Feature Testing

The functions in this section describe the basic capabilities of a particular display. Lisp programs can use them to adapt their behavior to what the display can do. For example, a program that ordinarily uses a popup menu could use the minibuffer if popup menus are not supported.

The optional argument *display* in these functions specifies which display to ask the question about. It can be a display name, a frame (which designates the display that frame is on), or `nil` (which refers to the selected frame's display, see Section 18.9 [Input Focus], page 358).

See Section 18.20 [Color Names], page 367, Section 18.21 [Text Terminal Colors], page 369, for other functions to obtain information about displays.

`display-popup-menus-p` **&optional** *display* [Function]
This function returns `t` if popup menus are supported on *display*, `nil` if not. Support for popup menus requires that the mouse be available, since the user cannot choose menu items without a mouse.

`display-graphic-p` **&optional** *display* [Function]
This function returns `t` if *display* is a graphic display capable of displaying several frames and several different fonts at once. This is true for displays that use a window system such as X, and false for text terminals.

`display-mouse-p` **&optional** *display* [Function]
This function returns `t` if *display* has a mouse available, `nil` if not.

`display-color-p` **&optional** *display* [Function]
This function returns `t` if the screen is a color screen. It used to be called `x-display-color-p`, and that name is still supported as an alias.

`display-grayscale-p` **&optional** *display* [Function]
This function returns `t` if the screen can display shades of gray. (All color displays can do this.)

`display-supports-face-attributes-p` *attributes* **&optional** *display* [Function]
This function returns non-`nil` if all the face attributes in *attributes* are supported (see Section 11.12.1 [Face Attributes], page 138).

The definition of 'supported' is somewhat heuristic, but basically means that a face containing all the attributes in *attributes*, when merged with the default face for display, can be represented in a way that's

1. different in appearance than the default face, and
2. 'close in spirit' to what the attributes specify, if not exact.

Point (2) implies that a `:weight black` attribute will be satisfied by any display that can display bold, as will `:foreground "yellow"` as long as some yellowish color can be displayed, but `:slant italic` will *not* be satisfied by the tty display code's automatic substitution of a 'dim' face for italic.

`display-selections-p` **&optional** *display* [Function]
This function returns `t` if *display* supports selections. Windowed displays normally support selections, but they may also be supported in some other cases.

`display-images-p` **&optional** *display* [Function]
This function returns `t` if *display* can display images. Windowed displays ought in principle to handle images, but some systems lack the support for that. On a display that does not support images, Emacs cannot display a tool bar.

`display-screens` **&optional** *display* [Function]
This function returns the number of screens associated with the display.

`display-pixel-height` **&optional** *display* [Function]
This function returns the height of the screen in pixels. On a character terminal, it gives the height in characters.

For graphical terminals, note that on "multi-monitor" setups this refers to the pixel width for all physical monitors associated with *display*. See Section 18.2 [Multiple Terminals], page 342.

`display-pixel-width` **&optional** *display* [Function]
This function returns the width of the screen in pixels. On a character terminal, it gives the width in characters.

For graphical terminals, note that on "multi-monitor" setups this refers to the pixel width for all physical monitors associated with *display*. See Section 18.2 [Multiple Terminals], page 342.

`display-mm-height` **&optional** *display* [Function]
This function returns the height of the screen in millimeters, or `nil` if Emacs cannot get that information.

`display-mm-width` **&optional** *display* [Function]
This function returns the width of the screen in millimeters, or `nil` if Emacs cannot get that information.

`display-mm-dimensions-alist` [User Option]
This variable allows the user to specify the dimensions of graphical displays returned by `display-mm-height` and `display-mm-width` in case the system provides incorrect values.

`display-backing-store` **&optional** *display* [Function]
This function returns the backing store capability of the display. Backing store means recording the pixels of windows (and parts of windows) that are not exposed, so that when exposed they can be displayed very quickly.

Values can be the symbols `always`, `when-mapped`, or `not-useful`. The function can also return `nil` when the question is inapplicable to a certain kind of display.

`display-save-under` **&optional** *display* [Function]
This function returns non-`nil` if the display supports the SaveUnder feature. That feature is used by pop-up windows to save the pixels they obscure, so that they can pop down quickly.

`display-planes` **&optional** *display* [Function]
This function returns the number of planes the display supports. This is typically the number of bits per pixel. For a tty display, it is log to base two of the number of colors supported.

`display-visual-class` **&optional** *display* [Function]
This function returns the visual class for the screen. The value is one of the symbols `static-gray` (a limited, unchangeable number of grays), `gray-scale` (a full range of grays), `static-color` (a limited, unchangeable number of colors), `pseudo-color` (a limited number of colors), `true-color` (a full range of colors), and `direct-color` (a full range of colors).

`display-color-cells` **&optional** *display* [Function]
This function returns the number of color cells the screen supports.

These functions obtain additional information specifically about X displays.

`x-server-version` **&optional** *display* [Function]
: This function returns the list of version numbers of the X server running the display. The value is a list of three integers: the major and minor version numbers of the X protocol, and the distributor-specific release number of the X server software itself.

`x-server-vendor` **&optional** *display* [Function]
: This function returns the "vendor" that provided the X server software (as a string). Really this means whoever distributes the X server.

  When the developers of X labeled software distributors as "vendors", they showed their false assumption that no system could ever be developed and distributed non-commercially.

# 19 International Character Set Support

Emacs supports a wide variety of international character sets, including European and Vietnamese variants of the Latin alphabet, as well as Cyrillic, Devanagari (for Hindi and Marathi), Ethiopic, Greek, Han (for Chinese and Japanese), Hangul (for Korean), Hebrew, IPA, Kannada, Lao, Malayalam, Tamil, Thai, Tibetan, and Vietnamese scripts. Emacs also supports various encodings of these characters that are used by other internationalized software, such as word processors and mailers.

Emacs allows editing text with international characters by supporting all the related activities:

- You can visit files with non-ASCII characters, save non-ASCII text, and pass non-ASCII text between Emacs and programs it invokes (such as compilers, spell-checkers, and mailers). Setting your language environment (see Section 19.3 [Language Environments], page 377) takes care of setting up the coding systems and other options for a specific language or culture. Alternatively, you can specify how Emacs should encode or decode text for each command; see Section 19.10 [Text Coding], page 386.
- You can display non-ASCII characters encoded by the various scripts. This works by using appropriate fonts on graphics displays (see Section 19.15 [Defining Fontsets], page 390), and by sending special codes to text displays (see Section 19.13 [Terminal Coding], page 388). If some characters are displayed incorrectly, refer to Section 19.17 [Undisplayable Characters], page 392, which describes possible problems and explains how to solve them.
- Characters from scripts whose natural ordering of text is from right to left are reordered for display (see Section 19.20 [Bidirectional Editing], page 394). These scripts include Arabic, Hebrew, Syriac, Thaana, and a few others.
- You can insert non-ASCII characters or search for them. To do that, you can specify an input method (see Section 19.5 [Select Input Method], page 380) suitable for your language, or use the default input method set up when you chose your language environment. If your keyboard can produce non-ASCII characters, you can select an appropriate keyboard coding system (see Section 19.13 [Terminal Coding], page 388), and Emacs will accept those characters. Latin-1 characters can also be input by using the `C-x 8` prefix, see Section 19.18 [Unibyte Mode], page 393.

  With the X Window System, your locale should be set to an appropriate value to make sure Emacs interprets keyboard input correctly; see Section 19.3 [Language Environments], page 377.

The rest of this chapter describes these issues in detail.

## 19.1 Introduction to International Character Sets

The users of international character sets and scripts have established many more-or-less standard coding systems for storing files. These coding systems are typically *multibyte*, meaning that sequences of two or more bytes are used to represent individual non-ASCII characters.

Internally, Emacs uses its own multibyte character encoding, which is a superset of the *Unicode* standard. This internal encoding allows characters from almost every known

script to be intermixed in a single buffer or string. Emacs translates between the multibyte character encoding and various other coding systems when reading and writing files, and when exchanging data with subprocesses.

The command `C-h h` (`view-hello-file`) displays the file '`etc/HELLO`', which illustrates various scripts by showing how to say "hello" in many languages. If some characters can't be displayed on your terminal, they appear as '`?`' or as hollow boxes (see Section 19.17 [Undisplayable Characters], page 392).

Keyboards, even in the countries where these character sets are used, generally don't have keys for all the characters in them. You can insert characters that your keyboard does not support, using `C-q` (`quoted-insert`) or `C-x 8 RET` (`insert-char`). See Section 4.1 [Inserting Text], page 58. Emacs also supports various *input methods*, typically one for each script or language, which make it easier to type characters in the script. See Section 19.4 [Input Methods], page 378.

The prefix key `C-x RET` is used for commands that pertain to multibyte characters, coding systems, and input methods.

The command `C-x =` (`what-cursor-position`) shows information about the character at point. In addition to the character position, which was described in Section 4.9 [Position Info], page 64, this command displays how the character is encoded. For instance, it displays the following line in the echo area for the character '`c`':

```
Char: c (99, #o143, #x63) point=28062 of 36168 (78%) column=53
```

The four values after '`Char:`' describe the character that follows point, first by showing it and then by giving its character code in decimal, octal and hex. For a non-ASCII multibyte character, these are followed by '`file`' and the character's representation, in hex, in the buffer's coding system, if that coding system encodes the character safely and with a single byte (see Section 19.6 [Coding Systems], page 381). If the character's encoding is longer than one byte, Emacs shows '`file ...`'.

As a special case, if the character lies in the range 128 (0200 octal) through 159 (0237 octal), it stands for a "raw" byte that does not correspond to any specific displayable character. Such a "character" lies within the `eight-bit-control` character set, and is displayed as an escaped octal character code. In this case, `C-x =` shows '`part of display ...`' instead of '`file`'.

With a prefix argument (`C-u C-x =`), this command displays a detailed description of the character in a window:

- The character set name, and the codes that identify the character within that character set; ASCII characters are identified as belonging to the `ascii` character set.
- The character's syntax and categories.
- The character's encodings, both internally in the buffer, and externally if you were to save the file.
- What keys to type to input the character in the current input method (if it supports the character).
- If you are running Emacs on a graphical display, the font name and glyph code for the character. If you are running Emacs on a text terminal, the code(s) sent to the terminal.

- The character's text properties (see Section "Text Properties" in *the Emacs Lisp Reference Manual*), including any non-default faces used to display the character, and any overlays containing it (see Section "Overlays" in *the same manual*).

Here's an example showing the Latin-1 character A with grave accent, in a buffer whose coding system is `utf-8-unix`:

```
               position: 1 of 1 (0%), column: 0
              character: À (displayed as À) (codepoint 192, #o300, #xc0)
    preferred charset: unicode (Unicode (ISO10646))
code point in charset: 0xC0
                 syntax: w       which means: word
               category: .:Base, L:Left-to-right (strong),
                         j:Japanese, l:Latin, v:Viet
          buffer code: #xC3 #x80
            file code: not encodable by coding system undecided-unix
              display: by this font (glyph code)
    xft:-unknown-DejaVu Sans Mono-normal-normal-
        normal-*-13-*-*-*-m-0-iso10646-1 (#x82)

Character code properties: customize what to show
  name: LATIN CAPITAL LETTER A WITH GRAVE
  old-name: LATIN CAPITAL LETTER A GRAVE
  general-category: Lu (Letter, Uppercase)
  decomposition: (65 768) ('A' '`')
```

## 19.2 Disabling Multibyte Characters

By default, Emacs starts in multibyte mode: it stores the contents of buffers and strings using an internal encoding that represents non-ASCII characters using multi-byte sequences. Multibyte mode allows you to use all the supported languages and scripts without limitations.

Under very special circumstances, you may want to disable multibyte character support, for a specific buffer. When multibyte characters are disabled in a buffer, we call that *unibyte mode*. In unibyte mode, each character in the buffer has a character code ranging from 0 through 255 (0377 octal); 0 through 127 (0177 octal) represent ASCII characters, and 128 (0200 octal) through 255 (0377 octal) represent non-ASCII characters.

To edit a particular file in unibyte representation, visit it using `find-file-literally`. See ⟨undefined⟩ [Visiting], page ⟨undefined⟩. You can convert a multibyte buffer to unibyte by saving it to a file, killing the buffer, and visiting the file again with `find-file-literally`. Alternatively, you can use `C-x RET c` (`universal-coding-system-argument`) and specify '`raw-text`' as the coding system with which to visit or save a file. See Section 19.10 [Text Coding], page 386. Unlike `find-file-literally`, finding a file as '`raw-text`' doesn't disable format conversion, uncompression, or auto mode selection.

Emacs normally loads Lisp files as multibyte. This includes the Emacs initialization file, '`.emacs`', and the initialization files of packages such as Gnus. However, you can specify unibyte loading for a particular Lisp file, by adding an entry '`coding: raw-text`' in a file local variables section. See Section 19.8 [Specify Coding], page 385. Then that file is always loaded as unibyte text. You can also load a Lisp file as unibyte, on any one occasion, by typing `C-x RET c raw-text RET` immediately before loading it.

The buffer-local variable `enable-multibyte-characters` is non-`nil` in multibyte buffers, and `nil` in unibyte ones. The mode line also indicates whether a buffer is multibyte

or not. See Section 1.3 [Mode Line], page 8. With a graphical display, in a multibyte buffer, the portion of the mode line that indicates the character set has a tooltip that (amongst other things) says that the buffer is multibyte. In a unibyte buffer, the character set indicator is absent. Thus, in a unibyte buffer (when using a graphical display) there is normally nothing before the indication of the visited file's end-of-line convention (colon, backslash, etc.), unless you are using an input method.

You can turn off multibyte support in a specific buffer by invoking the command `toggle-enable-multibyte-characters` in that buffer.

## 19.3 Language Environments

All supported character sets are supported in Emacs buffers whenever multibyte characters are enabled; there is no need to select a particular language in order to display its characters. However, it is important to select a *language environment* in order to set various defaults. Roughly speaking, the language environment represents a choice of preferred script rather than a choice of language.

The language environment controls which coding systems to recognize when reading text (see Section 19.7 [Recognize Coding], page 383). This applies to files, incoming mail, and any other text you read into Emacs. It may also specify the default coding system to use when you create a file. Each language environment also specifies a default input method.

To select a language environment, customize `current-language-environment` or use the command `M-x set-language-environment`. It makes no difference which buffer is current when you use this command, because the effects apply globally to the Emacs session. The supported language environments (see the variable `language-info-alist`) include:

> ASCII, Belarusian, Bengali, Brazilian Portuguese, Bulgarian, Cham, Chinese-BIG5, Chinese-CNS, Chinese-EUC-TW, Chinese-GB, Chinese-GBK, Chinese-GB18030, Croatian, Cyrillic-ALT, Cyrillic-ISO, Cyrillic-KOI8, Czech, Devanagari, Dutch, English, Esperanto, Ethiopic, French, Georgian, German, Greek, Gujarati, Hebrew, IPA, Italian, Japanese, Kannada, Khmer, Korean, Lao, Latin-1, Latin-2, Latin-3, Latin-4, Latin-5, Latin-6, Latin-7, Latin-8 (Celtic), Latin-9 (updated Latin-1 with the Euro sign), Latvian, Lithuanian, Malayalam, Oriya, Polish, Punjabi, Romanian, Russian, Sinhala, Slovak, Slovenian, Spanish, Swedish, TaiViet, Tajik, Tamil, Telugu, Thai, Tibetan, Turkish, UTF-8 (for a setup which prefers Unicode characters and files encoded in UTF-8), Ukrainian, Vietnamese, Welsh, and Windows-1255 (for a setup which prefers Cyrillic characters and files encoded in Windows-1255).

To display the script(s) used by your language environment on a graphical display, you need to have suitable fonts. See Section 19.14 [Fontsets], page 389, for more details about setting up your fonts.

Some operating systems let you specify the character-set locale you are using by setting the locale environment variables `LC_ALL`, `LC_CTYPE`, or `LANG`. (If more than one of these is set, the first one that is nonempty specifies your locale for this purpose.) During startup, Emacs looks up your character-set locale's name in the system locale alias table, matches its canonical name against entries in the value of the variables `locale-charset-language-names` and `locale-language-names` (the former overrides the latter), and selects the corresponding language environment if a match is found. It also adjusts the display

table and terminal coding system, the locale coding system, the preferred coding system as needed for the locale, and—last but not least—the way Emacs decodes non-ASCII characters sent by your keyboard.

If you modify the `LC_ALL`, `LC_CTYPE`, or `LANG` environment variables while running Emacs (by using `M-x setenv`), you may want to invoke the `set-locale-environment` function afterwards to readjust the language environment from the new locale.

The `set-locale-environment` function normally uses the preferred coding system established by the language environment to decode system messages. But if your locale matches an entry in the variable `locale-preferred-coding-systems`, Emacs uses the corresponding coding system instead. For example, if the locale '`ja_JP.PCK`' matches `japanese-shift-jis` in `locale-preferred-coding-systems`, Emacs uses that encoding even though it might normally use `japanese-iso-8bit`.

You can override the language environment chosen at startup with explicit use of the command `set-language-environment`, or with customization of `current-language-environment` in your init file.

To display information about the effects of a certain language environment *lang-env*, use the command `C-h L` *lang-env* `RET` (`describe-language-environment`). This tells you which languages this language environment is useful for, and lists the character sets, coding systems, and input methods that go with it. It also shows some sample text to illustrate scripts used in this language environment. If you give an empty input for *lang-env*, this command describes the chosen language environment.

You can customize any language environment with the normal hook `set-language-environment-hook`. The command `set-language-environment` runs that hook after setting up the new language environment. The hook functions can test for a specific language environment by checking the variable `current-language-environment`. This hook is where you should put non-default settings for specific language environments, such as coding systems for keyboard input and terminal output, the default input method, etc.

Before it starts to set up the new language environment, `set-language-environment` first runs the hook `exit-language-environment-hook`. This hook is useful for undoing customizations that were made with `set-language-environment-hook`. For instance, if you set up a special key binding in a specific language environment using `set-language-environment-hook`, you should set up `exit-language-environment-hook` to restore the normal binding for that key.

## 19.4 Input Methods

An *input method* is a kind of character conversion designed specifically for interactive input. In Emacs, typically each language has its own input method; sometimes several languages that use the same characters can share one input method. A few languages support several input methods.

The simplest kind of input method works by mapping ASCII letters into another alphabet; this allows you to use one other alphabet instead of ASCII. The Greek and Russian input methods work this way.

A more powerful technique is composition: converting sequences of characters into one letter. Many European input methods use composition to produce a single non-ASCII letter

from a sequence that consists of a letter followed by accent characters (or vice versa). For example, some methods convert the sequence `o ^` into a single accented letter. These input methods have no special commands of their own; all they do is compose sequences of printing characters.

The input methods for syllabic scripts typically use mapping followed by composition. The input methods for Thai and Korean work this way. First, letters are mapped into symbols for particular sounds or tone marks; then, sequences of these that make up a whole syllable are mapped into one syllable sign.

Chinese and Japanese require more complex methods. In Chinese input methods, first you enter the phonetic spelling of a Chinese word (in input method `chinese-py`, among others), or a sequence of portions of the character (input methods `chinese-4corner` and `chinese-sw`, and others). One input sequence typically corresponds to many possible Chinese characters. You select the one you mean using keys such as `C-f`, `C-b`, `C-n`, `C-p` (or the arrow keys), and digits, which have special meanings in this situation.

The possible characters are conceptually arranged in several rows, with each row holding up to 10 alternatives. Normally, Emacs displays just one row at a time, in the echo area; `(`*i*`/`*j*`)` appears at the beginning, to indicate that this is the *i*th row out of a total of *j* rows. Type `C-n` or `C-p` to display the next row or the previous row.

Type `C-f` and `C-b` to move forward and backward among the alternatives in the current row. As you do this, Emacs highlights the current alternative with a special color; type `C-SPC` to select the current alternative and use it as input. The alternatives in the row are also numbered; the number appears before the alternative. Typing a number selects the associated alternative of the current row and uses it as input.

TAB in these Chinese input methods displays a buffer showing all the possible characters at once; then clicking `Mouse-2` on one of them selects that alternative. The keys `C-f`, `C-b`, `C-n`, `C-p`, and digits continue to work as usual, but they do the highlighting in the buffer showing the possible characters, rather than in the echo area.

In Japanese input methods, first you input a whole word using phonetic spelling; then, after the word is in the buffer, Emacs converts it into one or more characters using a large dictionary. One phonetic spelling corresponds to a number of different Japanese words; to select one of them, use `C-n` and `C-p` to cycle through the alternatives.

Sometimes it is useful to cut off input method processing so that the characters you have just entered will not combine with subsequent characters. For example, in input method `latin-1-postfix`, the sequence `o ^` combines to form an ‘o’ with an accent. What if you want to enter them as separate characters?

One way is to type the accent twice; this is a special feature for entering the separate letter and accent. For example, `o ^ ^` gives you the two characters ‘o^’. Another way is to type another letter after the `o`—something that won't combine with that—and immediately delete it. For example, you could type `o o DEL ^` to get separate ‘o’ and ‘^’.

Another method, more general but not quite as easy to type, is to use `C-\ C-\` between two characters to stop them from combining. This is the command `C-\` (`toggle-input-method`) used twice.

`C-\ C-\` is especially useful inside an incremental search, because it stops waiting for more characters to combine, and starts searching for what you have already entered.

To find out how to input the character after point using the current input method, type `C-u C-x =`. See Section 4.9 [Position Info], page 64.

The variables `input-method-highlight-flag` and `input-method-verbose-flag` control how input methods explain what is happening. If `input-method-highlight-flag` is non-`nil`, the partial sequence is highlighted in the buffer (for most input methods—some disable this feature). If `input-method-verbose-flag` is non-`nil`, the list of possible characters to type next is displayed in the echo area (but not when you are in the minibuffer).

Another facility for typing characters not on your keyboard is by using `C-x 8 RET` (`insert-char`) to insert a single character based on its Unicode name or code-point; see Section 4.1 [Inserting Text], page 58.

## 19.5 Selecting an Input Method

`C-\` Enable or disable use of the selected input method (`toggle-input-method`).

`C-x RET C-\` *method* `RET`
: Select a new input method for the current buffer (`set-input-method`).

`C-h I` *method* `RET`
`C-h C-\` *method* `RET`
: Describe the input method *method* (`describe-input-method`). By default, it describes the current input method (if any). This description should give you the full details of how to use any particular input method.

`M-x list-input-methods`
: Display a list of all the supported input methods.

To choose an input method for the current buffer, use `C-x RET C-\` (`set-input-method`). This command reads the input method name from the minibuffer; the name normally starts with the language environment that it is meant to be used with. The variable `current-input-method` records which input method is selected.

Input methods use various sequences of ASCII characters to stand for non-ASCII characters. Sometimes it is useful to turn off the input method temporarily. To do this, type `C-\` (`toggle-input-method`). To reenable the input method, type `C-\` again.

If you type `C-\` and you have not yet selected an input method, it prompts you to specify one. This has the same effect as using `C-x RET C-\` to specify an input method.

When invoked with a numeric argument, as in `C-u C-\`, `toggle-input-method` always prompts you for an input method, suggesting the most recently selected one as the default.

Selecting a language environment specifies a default input method for use in various buffers. When you have a default input method, you can select it in the current buffer by typing `C-\`. The variable `default-input-method` specifies the default input method (`nil` means there is none).

In some language environments, which support several different input methods, you might want to use an input method different from the default chosen by `set-language-environment`. You can instruct Emacs to select a different default input method for a certain language environment, if you wish, by using `set-language-environment-hook` (see Section 19.3 [Language Environments], page 377). For example:

```
(defun my-chinese-setup ()
  "Set up my private Chinese environment."
  (if (equal current-language-environment "Chinese-GB")
      (setq default-input-method "chinese-tonepy")))
(add-hook 'set-language-environment-hook 'my-chinese-setup)
```

This sets the default input method to be `chinese-tonepy` whenever you choose a Chinese-GB language environment.

You can instruct Emacs to activate a certain input method automatically. For example:

```
(add-hook 'text-mode-hook
  (lambda () (set-input-method "german-prefix")))
```

This automatically activates the input method "german-prefix" in Text mode.

Some input methods for alphabetic scripts work by (in effect) remapping the keyboard to emulate various keyboard layouts commonly used for those scripts. How to do this remapping properly depends on your actual keyboard layout. To specify which layout your keyboard has, use the command `M-x quail-set-keyboard-layout`.

You can use the command `M-x quail-show-key` to show what key (or key sequence) to type in order to input the character following point, using the selected keyboard layout. The command `C-u C-x =` also shows that information, in addition to other information about the character.

`M-x list-input-methods` displays a list of all the supported input methods. The list gives information about each input method, including the string that stands for it in the mode line.

## 19.6 Coding Systems

Users of various languages have established many more-or-less standard coding systems for representing them. Emacs does not use these coding systems internally; instead, it converts from various coding systems to its own system when reading data, and converts the internal coding system to other coding systems when writing data. Conversion is possible in reading or writing files, in sending or receiving from the terminal, and in exchanging data with subprocesses.

Emacs assigns a name to each coding system. Most coding systems are used for one language, and the name of the coding system starts with the language name. Some coding systems are used for several languages; their names usually start with '`iso`'. There are also special coding systems, such as `no-conversion`, `raw-text`, and `emacs-internal`.

A special class of coding systems, collectively known as *codepages*, is designed to support text encoded by MS-Windows and MS-DOS software. The names of these coding systems are `cp`*`nnnn`*, where *nnnn* is a 3- or 4-digit number of the codepage. You can use these encodings just like any other coding system; for example, to visit a file encoded in codepage 850, type `C-x RET c cp850 RET C-x C-f` *`filename`* `RET`.

In addition to converting various representations of non-ASCII characters, a coding system can perform end-of-line conversion. Emacs handles three different conventions for how to separate lines in a file: newline ("unix"), carriage-return linefeed ("dos"), and just carriage-return ("mac").

`C-h C` *coding* `RET`
: Describe coding system *coding* (`describe-coding-system`).

`C-h C RET` Describe the coding systems currently in use.

`M-x list-coding-systems`
: Display a list of all the supported coding systems.

The command `C-h C` (`describe-coding-system`) displays information about particular coding systems, including the end-of-line conversion specified by those coding systems. You can specify a coding system name as the argument; alternatively, with an empty argument, it describes the coding systems currently selected for various purposes, both in the current buffer and as the defaults, and the priority list for recognizing coding systems (see Section 19.7 [Recognize Coding], page 383).

To display a list of all the supported coding systems, type `M-x list-coding-systems`. The list gives information about each coding system, including the letter that stands for it in the mode line (see Section 1.3 [Mode Line], page 8).

Each of the coding systems that appear in this list—except for `no-conversion`, which means no conversion of any kind—specifies how and whether to convert printing characters, but leaves the choice of end-of-line conversion to be decided based on the contents of each file. For example, if the file appears to use the sequence carriage-return linefeed to separate lines, DOS end-of-line conversion will be used.

Each of the listed coding systems has three variants, which specify exactly what to do for end-of-line conversion:

`...-unix` Don't do any end-of-line conversion; assume the file uses newline to separate lines. (This is the convention normally used on Unix and GNU systems, and Mac OS X.)

`...-dos` Assume the file uses carriage-return linefeed to separate lines, and do the appropriate conversion. (This is the convention normally used on Microsoft systems.[1])

`...-mac` Assume the file uses carriage-return to separate lines, and do the appropriate conversion. (This was the convention used on the Macintosh system prior to OS X.)

These variant coding systems are omitted from the `list-coding-systems` display for brevity, since they are entirely predictable. For example, the coding system `iso-latin-1` has variants `iso-latin-1-unix`, `iso-latin-1-dos` and `iso-latin-1-mac`.

The coding systems `unix`, `dos`, and `mac` are aliases for `undecided-unix`, `undecided-dos`, and `undecided-mac`, respectively. These coding systems specify only the end-of-line conversion, and leave the character code conversion to be deduced from the text itself.

The coding system `raw-text` is good for a file which is mainly ASCII text, but may contain byte values above 127 that are not meant to encode non-ASCII characters. With `raw-text`, Emacs copies those byte values unchanged, and sets `enable-multibyte-characters` to `nil` in the current buffer so that they will be interpreted properly. `raw-text` handles

[1] It is also specified for MIME '`text/*`' bodies and in other network transport contexts. It is different from the SGML reference syntax record-start/record-end format, which Emacs doesn't support directly.

end-of-line conversion in the usual way, based on the data encountered, and has the usual three variants to specify the kind of end-of-line conversion to use.

In contrast, the coding system `no-conversion` specifies no character code conversion at all—none for non-ASCII byte values and none for end of line. This is useful for reading or writing binary files, tar files, and other files that must be examined verbatim. It, too, sets `enable-multibyte-characters` to `nil`.

The easiest way to edit a file with no conversion of any kind is with the `M-x find-file-literally` command. This uses `no-conversion`, and also suppresses other Emacs features that might convert the file contents before you see them. See ⟨undefined⟩ [Visiting], page ⟨undefined⟩.

The coding system `emacs-internal` (or `utf-8-emacs`, which is equivalent) means that the file contains non-ASCII characters stored with the internal Emacs encoding. This coding system handles end-of-line conversion based on the data encountered, and has the usual three variants to specify the kind of end-of-line conversion.

## 19.7 Recognizing Coding Systems

Whenever Emacs reads a given piece of text, it tries to recognize which coding system to use. This applies to files being read, output from subprocesses, text from X selections, etc. Emacs can select the right coding system automatically most of the time—once you have specified your preferences.

Some coding systems can be recognized or distinguished by which byte sequences appear in the data. However, there are coding systems that cannot be distinguished, not even potentially. For example, there is no way to distinguish between Latin-1 and Latin-2; they use the same byte values with different meanings.

Emacs handles this situation by means of a priority list of coding systems. Whenever Emacs reads a file, if you do not specify the coding system to use, Emacs checks the data against each coding system, starting with the first in priority and working down the list, until it finds a coding system that fits the data. Then it converts the file contents assuming that they are represented in this coding system.

The priority list of coding systems depends on the selected language environment (see Section 19.3 [Language Environments], page 377). For example, if you use French, you probably want Emacs to prefer Latin-1 to Latin-2; if you use Czech, you probably want Latin-2 to be preferred. This is one of the reasons to specify a language environment.

However, you can alter the coding system priority list in detail with the command `M-x prefer-coding-system`. This command reads the name of a coding system from the minibuffer, and adds it to the front of the priority list, so that it is preferred to all others. If you use this command several times, each use adds one element to the front of the priority list.

If you use a coding system that specifies the end-of-line conversion type, such as `iso-8859-1-dos`, what this means is that Emacs should attempt to recognize `iso-8859-1` with priority, and should use DOS end-of-line conversion when it does recognize `iso-8859-1`.

Sometimes a file name indicates which coding system to use for the file. The variable `file-coding-system-alist` specifies this correspondence. There is a special function `modify-coding-system-alist` for adding elements to this list. For example, to read and

write all '`.txt`' files using the coding system `chinese-iso-8bit`, you can execute this Lisp expression:

```
(modify-coding-system-alist 'file "\\.txt\\'" 'chinese-iso-8bit)
```

The first argument should be `file`, the second argument should be a regular expression that determines which files this applies to, and the third argument says which coding system to use for these files.

Emacs recognizes which kind of end-of-line conversion to use based on the contents of the file: if it sees only carriage-returns, or only carriage-return linefeed sequences, then it chooses the end-of-line conversion accordingly. You can inhibit the automatic use of end-of-line conversion by setting the variable `inhibit-eol-conversion` to non-`nil`. If you do that, DOS-style files will be displayed with the '`^M`' characters visible in the buffer; some people prefer this to the more subtle '`(DOS)`' end-of-line type indication near the left edge of the mode line (see Section 1.3 [Mode Line], page 8).

By default, the automatic detection of coding system is sensitive to escape sequences. If Emacs sees a sequence of characters that begin with an escape character, and the sequence is valid as an ISO-2022 code, that tells Emacs to use one of the ISO-2022 encodings to decode the file.

However, there may be cases that you want to read escape sequences in a file as is. In such a case, you can set the variable `inhibit-iso-escape-detection` to non-`nil`. Then the code detection ignores any escape sequences, and never uses an ISO-2022 encoding. The result is that all escape sequences become visible in the buffer.

The default value of `inhibit-iso-escape-detection` is `nil`. We recommend that you not change it permanently, only for one specific operation. That's because some Emacs Lisp source files in the Emacs distribution contain non-ASCII characters encoded in the coding system `iso-2022-7bit`, and they won't be decoded correctly when you visit those files if you suppress the escape sequence detection.

The variables `auto-coding-alist` and `auto-coding-regexp-alist` are the strongest way to specify the coding system for certain patterns of file names, or for files containing certain patterns, respectively. These variables even override '`-*-coding:-*-`' tags in the file itself (see Section 19.8 [Specify Coding], page 385). For example, Emacs uses `auto-coding-alist` for tar and archive files, to prevent it from being confused by a '`-*-coding:-*-`' tag in a member of the archive and thinking it applies to the archive file as a whole.

Another way to specify a coding system is with the variable `auto-coding-functions`. For example, one of the builtin `auto-coding-functions` detects the encoding for XML files. Unlike the previous two, this variable does not override any '`-*-coding:-*-`' tag.

When you get new mail in Rmail, each message is translated automatically from the coding system it is written in, as if it were a separate file. This uses the priority list of coding systems that you have specified. If a MIME message specifies a character set, Rmail obeys that specification. For reading and saving Rmail files themselves, Emacs uses the coding system specified by the variable `rmail-file-coding-system`. The default value is `nil`, which means that Rmail files are not translated (they are read and written in the Emacs internal character code).

## 19.8 Specifying a File's Coding System

If Emacs recognizes the encoding of a file incorrectly, you can reread the file using the correct coding system with `C-x RET r` (`revert-buffer-with-coding-system`). This command prompts for the coding system to use. To see what coding system Emacs actually used to decode the file, look at the coding system mnemonic letter near the left edge of the mode line (see Section 1.3 [Mode Line], page 8), or type `C-h C` (`describe-coding-system`).

You can specify the coding system for a particular file in the file itself, using the '`-*-...-*-`' construct at the beginning, or a local variables list at the end (see Section 33.2.4 [File Variables], page 698). You do this by defining a value for the "variable" named `coding`. Emacs does not really have a variable `coding`; instead of setting a variable, this uses the specified coding system for the file. For example, '`-*-mode: C; coding: latin-1;-*-`' specifies use of the Latin-1 coding system, as well as C mode. When you specify the coding explicitly in the file, that overrides `file-coding-system-alist`.

## 19.9 Choosing Coding Systems for Output

Once Emacs has chosen a coding system for a buffer, it stores that coding system in `buffer-file-coding-system`. That makes it the default for operations that write from this buffer into a file, such as `save-buffer` and `write-region`. You can specify a different coding system for further file output from the buffer using `set-buffer-file-coding-system` (see Section 19.10 [Text Coding], page 386).

You can insert any character Emacs supports into any Emacs buffer, but most coding systems can only handle a subset of these characters. Therefore, it's possible that the characters you insert cannot be encoded with the coding system that will be used to save the buffer. For example, you could visit a text file in Polish, encoded in `iso-8859-2`, and add some Russian words to it. When you save that buffer, Emacs cannot use the current value of `buffer-file-coding-system`, because the characters you added cannot be encoded by that coding system.

When that happens, Emacs tries the most-preferred coding system (set by `M-x prefer-coding-system` or `M-x set-language-environment`). If that coding system can safely encode all of the characters in the buffer, Emacs uses it, and stores its value in `buffer-file-coding-system`. Otherwise, Emacs displays a list of coding systems suitable for encoding the buffer's contents, and asks you to choose one of those coding systems.

If you insert the unsuitable characters in a mail message, Emacs behaves a bit differently. It additionally checks whether the most-preferred coding system is recommended for use in MIME messages; if not, it informs you of this fact and prompts you for another coding system. This is so you won't inadvertently send a message encoded in a way that your recipient's mail software will have difficulty decoding. (You can still use an unsuitable coding system if you enter its name at the prompt.)

When you send a mail message (see Chapter 29 [Sending Mail], page 624), Emacs has four different ways to determine the coding system to use for encoding the message text. It tries the buffer's own value of `buffer-file-coding-system`, if that is non-`nil`. Otherwise, it uses the value of `sendmail-coding-system`, if that is non-`nil`. The third way is to use the default coding system for new files, which is controlled by your choice of language environment, if that is non-`nil`. If all of these three values are `nil`, Emacs encodes outgoing mail using the Latin-1 coding system.

## 19.10 Specifying a Coding System for File Text

In cases where Emacs does not automatically choose the right coding system for a file's contents, you can use these commands to specify one:

`C-x RET f` *coding* `RET`
: Use coding system *coding* to save or revisit the file in the current buffer (`set-buffer-file-coding-system`).

`C-x RET c` *coding* `RET`
: Specify coding system *coding* for the immediately following command (`universal-coding-system-argument`).

`C-x RET r` *coding* `RET`
: Revisit the current file using the coding system *coding* (`revert-buffer-with-coding-system`).

`M-x recode-region RET` *right* `RET` *wrong* `RET`
: Convert a region that was decoded using coding system *wrong*, decoding it using coding system *right* instead.

The command `C-x RET f` (`set-buffer-file-coding-system`) sets the file coding system for the current buffer (i.e., the coding system to use when saving or reverting the file). You specify which coding system using the minibuffer. You can also invoke this command by clicking with `Mouse-3` on the coding system indicator in the mode line (see Section 1.3 [Mode Line], page 8).

If you specify a coding system that cannot handle all the characters in the buffer, Emacs will warn you about the troublesome characters, and ask you to choose another coding system, when you try to save the buffer (see Section 19.9 [Output Coding], page 385).

You can also use this command to specify the end-of-line conversion (see Section 19.6 [Coding Systems], page 381) for encoding the current buffer. For example, `C-x RET f dos RET` will cause Emacs to save the current buffer's text with DOS-style carriage-return linefeed line endings.

Another way to specify the coding system for a file is when you visit the file. First use the command `C-x RET c` (`universal-coding-system-argument`); this command uses the minibuffer to read a coding system name. After you exit the minibuffer, the specified coding system is used for *the immediately following command*.

So if the immediately following command is `C-x C-f`, for example, it reads the file using that coding system (and records the coding system for when you later save the file). Or if the immediately following command is `C-x C-w`, it writes the file using that coding system. When you specify the coding system for saving in this way, instead of with `C-x RET f`, there is no warning if the buffer contains characters that the coding system cannot handle.

Other file commands affected by a specified coding system include `C-x i` and `C-x C-v`, as well as the other-window variants of `C-x C-f`. `C-x RET c` also affects commands that start subprocesses, including `M-x shell` (see Section 31.3 [Shell], page 655). If the immediately following command does not use the coding system, then `C-x RET c` ultimately has no effect.

An easy way to visit a file with no conversion is with the `M-x find-file-literally` command. See ⟨undefined⟩ [Visiting], page ⟨undefined⟩.

The default value of the variable `buffer-file-coding-system` specifies the choice of coding system to use when you create a new file. It applies when you find a new file, and when you create a buffer and then save it in a file. Selecting a language environment typically sets this variable to a good choice of default coding system for that language environment.

If you visit a file with a wrong coding system, you can correct this with `C-x RET r` (`revert-buffer-with-coding-system`). This visits the current file again, using a coding system you specify.

If a piece of text has already been inserted into a buffer using the wrong coding system, you can redo the decoding of it using `M-x recode-region`. This prompts you for the proper coding system, then for the wrong coding system that was actually used, and does the conversion. It first encodes the region using the wrong coding system, then decodes it again using the proper coding system.

## 19.11 Coding Systems for Interprocess Communication

This section explains how to specify coding systems for use in communication with other processes.

`C-x RET x` *coding* `RET`
: Use coding system *coding* for transferring selections to and from other graphical applications (`set-selection-coding-system`).

`C-x RET X` *coding* `RET`
: Use coding system *coding* for transferring *one* selection—the next one—to or from another graphical application (`set-next-selection-coding-system`).

`C-x RET p` *input-coding* `RET` *output-coding* `RET`
: Use coding systems *input-coding* and *output-coding* for subprocess input and output in the current buffer (`set-buffer-process-coding-system`).

The command `C-x RET x` (`set-selection-coding-system`) specifies the coding system for sending selected text to other windowing applications, and for receiving the text of selections made in other applications. This command applies to all subsequent selections, until you override it by using the command again. The command `C-x RET X` (`set-next-selection-coding-system`) specifies the coding system for the next selection made in Emacs or read by Emacs.

The variable `x-select-request-type` specifies the data type to request from the X Window System for receiving text selections from other applications. If the value is `nil` (the default), Emacs tries `UTF8_STRING` and `COMPOUND_TEXT`, in this order, and uses various heuristics to choose the more appropriate of the two results; if none of these succeed, Emacs falls back on `STRING`. If the value of `x-select-request-type` is one of the symbols `COMPOUND_TEXT`, `UTF8_STRING`, `STRING`, or `TEXT`, Emacs uses only that request type. If the value is a list of some of these symbols, Emacs tries only the request types in the list, in order, until one of them succeeds, or until the list is exhausted.

The command `C-x RET p` (`set-buffer-process-coding-system`) specifies the coding system for input and output to a subprocess. This command applies to the current buffer; normally, each subprocess has its own buffer, and thus you can use this command to specify

translation to and from a particular subprocess by giving the command in the corresponding buffer.

You can also use `C-x RET c` (`universal-coding-system-argument`) just before the command that runs or starts a subprocess, to specify the coding system for communicating with that subprocess. See Section 19.10 [Text Coding], page 386.

The default for translation of process input and output depends on the current language environment.

The variable `locale-coding-system` specifies a coding system to use when encoding and decoding system strings such as system error messages and `format-time-string` formats and time stamps. That coding system is also used for decoding non-ASCII keyboard input on the X Window System. You should choose a coding system that is compatible with the underlying system's text representation, which is normally specified by one of the environment variables `LC_ALL`, `LC_CTYPE`, and `LANG`. (The first one, in the order specified above, whose value is nonempty is the one that determines the text representation.)

## 19.12 Coding Systems for File Names

`C-x RET F` *coding* `RET`
: Use coding system *coding* for encoding and decoding file names (`set-file-name-coding-system`).

The command `C-x RET F` (`set-file-name-coding-system`) specifies a coding system to use for encoding file *names*. It has no effect on reading and writing the *contents* of files.

In fact, all this command does is set the value of the variable `file-name-coding-system`. If you set the variable to a coding system name (as a Lisp symbol or a string), Emacs encodes file names using that coding system for all file operations. This makes it possible to use non-ASCII characters in file names—or, at least, those non-ASCII characters that the specified coding system can encode.

If `file-name-coding-system` is `nil`, Emacs uses a default coding system determined by the selected language environment, and stored in the `default-file-name-coding-system` variable. In the default language environment, non-ASCII characters in file names are not encoded specially; they appear in the file system using the internal Emacs representation.

**Warning:** if you change `file-name-coding-system` (or the language environment) in the middle of an Emacs session, problems can result if you have already visited files whose names were encoded using the earlier coding system and cannot be encoded (or are encoded differently) under the new coding system. If you try to save one of these buffers under the visited file name, saving may use the wrong file name, or it may encounter an error. If such a problem happens, use `C-x C-w` to specify a new file name for that buffer.

If a mistake occurs when encoding a file name, use the command `M-x recode-file-name` to change the file name's coding system. This prompts for an existing file name, its old coding system, and the coding system to which you wish to convert.

## 19.13 Coding Systems for Terminal I/O

`C-x RET t` *coding* `RET`
: Use coding system *coding* for terminal output (`set-terminal-coding-system`).

`C-x RET k` *coding* `RET`
: Use coding system *coding* for keyboard input (`set-keyboard-coding-system`).

The command `C-x RET t` (`set-terminal-coding-system`) specifies the coding system for terminal output. If you specify a character code for terminal output, all characters output to the terminal are translated into that coding system.

This feature is useful for certain character-only terminals built to support specific languages or character sets—for example, European terminals that support one of the ISO Latin character sets. You need to specify the terminal coding system when using multibyte text, so that Emacs knows which characters the terminal can actually handle.

By default, output to the terminal is not translated at all, unless Emacs can deduce the proper coding system from your terminal type or your locale specification (see Section 19.3 [Language Environments], page 377).

The command `C-x RET k` (`set-keyboard-coding-system`), or the variable `keyboard-coding-system`, specifies the coding system for keyboard input. Character-code translation of keyboard input is useful for terminals with keys that send non-ASCII graphic characters—for example, some terminals designed for ISO Latin-1 or subsets of it.

By default, keyboard input is translated based on your system locale setting. If your terminal does not really support the encoding implied by your locale (for example, if you find it inserts a non-ASCII character if you type `M-i`), you will need to set `keyboard-coding-system` to `nil` to turn off encoding. You can do this by putting

```
(set-keyboard-coding-system nil)
```

in your init file.

There is a similarity between using a coding system translation for keyboard input, and using an input method: both define sequences of keyboard input that translate into single characters. However, input methods are designed to be convenient for interactive use by humans, and the sequences that are translated are typically sequences of ASCII printing characters. Coding systems typically translate sequences of non-graphic characters.

## 19.14 Fontsets

A font typically defines shapes for a single alphabet or script. Therefore, displaying the entire range of scripts that Emacs supports requires a collection of many fonts. In Emacs, such a collection is called a *fontset*. A fontset is defined by a list of font specifications, each assigned to handle a range of character codes, and may fall back on another fontset for characters that are not covered by the fonts it specifies.

Each fontset has a name, like a font. However, while fonts are stored in the system and the available font names are defined by the system, fontsets are defined within Emacs itself. Once you have defined a fontset, you can use it within Emacs by specifying its name, anywhere that you could use a single font. Of course, Emacs fontsets can use only the fonts that the system supports. If some characters appear on the screen as empty boxes or hex codes, this means that the fontset in use for them has no font for those characters. In this case, or if the characters are shown, but not as well as you would like, you may need to

install extra fonts. Your operating system may have optional fonts that you can install; or you can install the GNU Intlfonts package, which includes fonts for most supported scripts.[2]

Emacs creates three fontsets automatically: the *standard fontset*, the *startup fontset* and the *default fontset*. The default fontset is most likely to have fonts for a wide variety of non-ASCII characters, and is the default fallback for the other two fontsets, and if you set a default font rather than fontset. However, it does not specify font family names, so results can be somewhat random if you use it directly. You can specify use of a particular fontset by starting Emacs with the '`-fn`' option. For example,

```
emacs -fn fontset-standard
```

You can also specify a fontset with the '`Font`' resource (see Appendix F [X Resources], page 769).

If no fontset is specified for use, then Emacs uses an ASCII font, with '`fontset-default`' as a fallback for characters the font does not cover. The standard fontset is only used if explicitly requested, despite its name.

A fontset does not necessarily specify a font for every character code. If a fontset specifies no font for a certain character, or if it specifies a font that does not exist on your system, then it cannot display that character properly. It will display that character as a hex code or thin space or an empty box instead. (See ⟨undefined⟩ [glyphless characters], page ⟨undefined⟩, for details.)

## 19.15 Defining fontsets

When running on X, Emacs creates a standard fontset automatically according to the value of `standard-fontset-spec`. This fontset's name is

```
-*-fixed-medium-r-normal-*-16-*-*-*-*-*-fontset-standard
```

or just '`fontset-standard`' for short.

On GNUstep and Mac OS X, the standard fontset is created using the value of `ns-standard-fontset-spec`, and on MS Windows it is created using the value of `w32-standard-fontset-spec`.

Bold, italic, and bold-italic variants of the standard fontset are created automatically. Their names have '`bold`' instead of '`medium`', or '`i`' instead of '`r`', or both.

Emacs generates a fontset automatically, based on any default ASCII font that you specify with the '`Font`' resource or the '`-fn`' argument, or the default font that Emacs found when it started. This is the *startup fontset* and its name is `fontset-startup`. It does this by replacing the *charset_registry* field with '`fontset`', and replacing *charset_encoding* field with '`startup`', then using the resulting string to specify a fontset.

For instance, if you start Emacs with a font of this form,

```
emacs -fn "*courier-medium-r-normal--14-140-*-iso8859-1"
```

Emacs generates the following fontset and uses it for the initial X window frame:

---

[2] If you run Emacs on X, you may need to inform the X server about the location of the newly installed fonts with commands such as:

```
xset fp+ /usr/local/share/emacs/fonts
xset fp rehash
```

```
-*-courier-medium-r-normal-*-14-140-*-*-*-*-fontset-startup
```

The startup fontset will use the font that you specify, or a variant with a different registry and encoding, for all the characters that are supported by that font, and fallback on '`fontset-default`' for other characters.

With the X resource '`Emacs.Font`', you can specify a fontset name just like an actual font name. But be careful not to specify a fontset name in a wildcard resource like '`Emacs*Font`'—that wildcard specification matches various other resources, such as for menus, and menus cannot handle fontsets. See Appendix F [X Resources], page 769.

You can specify additional fontsets using X resources named '`Fontset-`*n*', where *n* is an integer starting from 0. The resource value should have this form:

*fontpattern*, [*charset*:*font*]...

*fontpattern* should have the form of a standard X font name (see the previous fontset-startup example), except for the last two fields. They should have the form '`fontset-`*alias*'.

The fontset has two names, one long and one short. The long name is *fontpattern*. The short name is '`fontset-`*alias*'. You can refer to the fontset by either name.

The construct '*charset*:*font*' specifies which font to use (in this fontset) for one particular character set. Here, *charset* is the name of a character set, and *font* is the font to use for that character set. You can use this construct any number of times in defining one fontset.

For the other character sets, Emacs chooses a font based on *fontpattern*. It replaces '`fontset-`*alias*' with values that describe the character set. For the ASCII character font, '`fontset-`*alias*' is replaced with '`ISO8859-1`'.

In addition, when several consecutive fields are wildcards, Emacs collapses them into a single wildcard. This is to prevent use of auto-scaled fonts. Fonts made by scaling larger fonts are not usable for editing, and scaling a smaller font is not also useful, because it is better to use the smaller font in its own size, which is what Emacs does.

Thus if *fontpattern* is this,

```
-*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24
```

the font specification for ASCII characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-ISO8859-1
```

and the font specification for Chinese GB2312 characters would be this:

```
-*-fixed-medium-r-normal-*-24-*-gb2312*-*
```

You may not have any Chinese font matching the above font specification. Most X distributions include only Chinese fonts that have '`song ti`' or '`fangsong ti`' in the *family* field. In such a case, '`Fontset-`*n*' can be specified as:

```
Emacs.Fontset-0: -*-fixed-medium-r-normal-*-24-*-*-*-*-*-fontset-24,\
        chinese-gb2312:-*-*-medium-r-normal-*-24-*-gb2312*-*
```

Then, the font specifications for all but Chinese GB2312 characters have '`fixed`' in the *family* field, and the font specification for Chinese GB2312 characters has a wild card '`*`' in the *family* field.

The function that processes the fontset resource value to create the fontset is called `create-fontset-from-fontset-spec`. You can also call this function explicitly to create a fontset.

See ⟨undefined⟩ [Fonts], page ⟨undefined⟩, for more information about font naming.

## 19.16 Modifying Fontsets

Fontsets do not always have to be created from scratch. If only minor changes are required it may be easier to modify an existing fontset. Modifying '`fontset-default`' will also affect other fontsets that use it as a fallback, so can be an effective way of fixing problems with the fonts that Emacs chooses for a particular script.

Fontsets can be modified using the function `set-fontset-font`, specifying a character, a charset, a script, or a range of characters to modify the font for, and a font specification for the font to be used. Some examples are:

```
;; Use Liberation Mono for latin-3 charset.
(set-fontset-font "fontset-default" 'iso-8859-3
                  "Liberation Mono")

;; Prefer a big5 font for han characters
(set-fontset-font "fontset-default"
                  'han (font-spec :registry "big5")
                  nil 'prepend)

;; Use DejaVu Sans Mono as a fallback in fontset-startup
;; before resorting to fontset-default.
(set-fontset-font "fontset-startup" nil "DejaVu Sans Mono"
                  nil 'append)

;; Use MyPrivateFont for the Unicode private use area.
(set-fontset-font "fontset-default"  '(#xe000 . #xf8ff)
                  "MyPrivateFont")
```

## 19.17 Undisplayable Characters

There may be some non-ASCII characters that your terminal cannot display. Most text terminals support just a single character set (use the variable `default-terminal-coding-system` to tell Emacs which one, Section 19.13 [Terminal Coding], page 388); characters that can't be encoded in that coding system are displayed as '`?`' by default.

Graphical displays can display a broader range of characters, but you may not have fonts installed for all of them; characters that have no font appear as a hollow box.

If you use Latin-1 characters but your terminal can't display Latin-1, you can arrange to display mnemonic ASCII sequences instead, e.g., '`"o`' for o-umlaut. Load the library '`iso-ascii`' to do this.

If your terminal can display Latin-1, you can display characters from other European character sets using a mixture of equivalent Latin-1 characters and ASCII mnemonics. Customize the variable `latin1-display` to enable this. The mnemonic ASCII sequences mostly correspond to those of the prefix input methods.

## 19.18 Unibyte Editing Mode

The ISO 8859 Latin-*n* character sets define character codes in the range 0240 to 0377 octal (160 to 255 decimal) to handle the accented letters and punctuation needed by various European languages (and some non-European ones). Note that Emacs considers bytes with codes in this range as raw bytes, not as characters, even in a unibyte buffer, i.e., if you disable multibyte characters. However, Emacs can still handle these character codes as if they belonged to *one* of the single-byte character sets at a time. To specify *which* of these codes to use, invoke `M-x set-language-environment` and specify a suitable language environment such as '`Latin-n`'.

For more information about unibyte operation, see Section 19.2 [Disabling Multibyte], page 376.

Emacs can also display bytes in the range 160 to 255 as readable characters, provided the terminal or font in use supports them. This works automatically. On a graphical display, Emacs can also display single-byte characters through fontsets, in effect by displaying the equivalent multibyte characters according to the current language environment. To request this, set the variable `unibyte-display-via-language-environment` to a non-`nil` value. Note that setting this only affects how these bytes are displayed, but does not change the fundamental fact that Emacs treats them as raw bytes, not as characters.

If your terminal does not support display of the Latin-1 character set, Emacs can display these characters as ASCII sequences which at least give you a clear idea of what the characters are. To do this, load the library `iso-ascii`. Similar libraries for other Latin-*n* character sets could be implemented, but have not been so far.

Normally non-ISO-8859 characters (decimal codes between 128 and 159 inclusive) are displayed as octal escapes. You can change this for non-standard "extended" versions of ISO-8859 character sets by using the function `standard-display-8bit` in the `disp-table` library.

There are two ways to input single-byte non-ASCII characters:

- You can use an input method for the selected language environment. See Section 19.4 [Input Methods], page 378. When you use an input method in a unibyte buffer, the non-ASCII character you specify with it is converted to unibyte.
- If your keyboard can generate character codes 128 (decimal) and up, representing non-ASCII characters, you can type those character codes directly.

  On a graphical display, you should not need to do anything special to use these keys; they should simply work. On a text terminal, you should use the command `M-x set-keyboard-coding-system` or customize the variable `keyboard-coding-system` to specify which coding system your keyboard uses (see Section 19.13 [Terminal Coding], page 388). Enabling this feature will probably require you to use `ESC` to type Meta characters; however, on a console terminal or in `xterm`, you can arrange for Meta to be converted to `ESC` and still be able type 8-bit characters present directly on the keyboard or using `Compose` or `AltGr` keys. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.
- For Latin-1 only, you can use the key `C-x 8` as a "compose character" prefix for entry of non-ASCII Latin-1 printing characters. `C-x 8` is good for insertion (in the minibuffer as well as other buffers), for searching, and in any other context where a key sequence is allowed.

`C-x 8` works by loading the `iso-transl` library. Once that library is loaded, the ALT modifier key, if the keyboard has one, serves the same purpose as `C-x 8`: use ALT together with an accent character to modify the following letter. In addition, if the keyboard has keys for the Latin-1 "dead accent characters", they too are defined to compose with the following character, once `iso-transl` is loaded.

Use `C-x 8 C-h` to list all the available `C-x 8` translations.

## 19.19 Charsets

In Emacs, *charset* is short for "character set". Emacs supports most popular charsets (such as `ascii`, `iso-8859-1`, `cp1250`, `big5`, and `unicode`), in addition to some charsets of its own (such as `emacs`, `unicode-bmp`, and `eight-bit`). All supported characters belong to one or more charsets.

Emacs normally "does the right thing" with respect to charsets, so that you don't have to worry about them. However, it is sometimes helpful to know some of the underlying details about charsets.

One example is font selection (see ⟨undefined⟩ [Fonts], page ⟨undefined⟩). Each language environment (see Section 19.3 [Language Environments], page 377) defines a "priority list" for the various charsets. When searching for a font, Emacs initially attempts to find one that can display the highest-priority charsets. For instance, in the Japanese language environment, the charset `japanese-jisx0208` has the highest priority, so Emacs tries to use a font whose `registry` property is '`JISX0208.1983-0`'.

There are two commands that can be used to obtain information about charsets. The command `M-x list-charset-chars` prompts for a charset name, and displays all the characters in that character set. The command `M-x describe-character-set` prompts for a charset name, and displays information about that charset, including its internal representation within Emacs.

`M-x list-character-sets` displays a list of all supported charsets. The list gives the names of charsets and additional information to identity each charset; see the International Register of Coded Character Sets for more details. In this list, charsets are divided into two categories: *normal charsets* are listed first, followed by *supplementary charsets*. A supplementary charset is one that is used to define another charset (as a parent or a subset), or to provide backward-compatibility for older Emacs versions.

To find out which charset a character in the buffer belongs to, put point before it and type `C-u C-x =` (see Section 19.1 [International Chars], page 374).

## 19.20 Bidirectional Editing

Emacs supports editing text written in scripts, such as Arabic and Hebrew, whose natural ordering of horizontal text for display is from right to left. However, digits and Latin text embedded in these scripts are still displayed left to right. It is also not uncommon to have small portions of text in Arabic or Hebrew embedded in an otherwise Latin document; e.g., as comments and strings in a program source file. For these reasons, text that uses these scripts is actually *bidirectional*: a mixture of runs of left-to-right and right-to-left characters.

This section describes the facilities and options provided by Emacs for editing bidirectional text.

Emacs stores right-to-left and bidirectional text in the so-called *logical* (or *reading*) order: the buffer or string position of the first character you read precedes that of the next character. Reordering of bidirectional text into the *visual* order happens at display time. As result, character positions no longer increase monotonically with their positions on display. Emacs implements the Unicode Bidirectional Algorithm described in the Unicode Standard Annex #9, for reordering of bidirectional text for display.

The buffer-local variable `bidi-display-reordering` controls whether text in the buffer is reordered for display. If its value is non-`nil`, Emacs reorders characters that have right-to-left directionality when they are displayed. The default value is `t`.

Each paragraph of bidirectional text can have its own *base direction*, either right-to-left or left-to-right. (Paragraph boundaries are empty lines, i.e., lines consisting entirely of whitespace characters.) Text in left-to-right paragraphs begins on the screen at the left margin of the window and is truncated or continued when it reaches the right margin. By contrast, text in right-to-left paragraphs is displayed starting at the right margin and is continued or truncated at the left margin.

Emacs determines the base direction of each paragraph dynamically, based on the text at the beginning of the paragraph. However, sometimes a buffer may need to force a certain base direction for its paragraphs. The variable `bidi-paragraph-direction`, if non-`nil`, disables the dynamic determination of the base direction, and instead forces all paragraphs in the buffer to have the direction specified by its buffer-local value. The value can be either `right-to-left` or `left-to-right`. Any other value is interpreted as `nil`.

Alternatively, you can control the base direction of a paragraph by inserting special formatting characters in front of the paragraph. The special character `RIGHT-TO-LEFT MARK`, or RLM, forces the right-to-left direction on the following paragraph, while `LEFT-TO-RIGHT MARK`, or LRM forces the left-to-right direction. (You can use `C-x 8 RET` to insert these characters.) In a GUI session, the LRM and RLM characters display as very thin blank characters; on text terminals they display as blanks.

Because characters are reordered for display, Emacs commands that operate in the logical order or on stretches of buffer positions may produce unusual effects. For example, `C-f` and `C-b` commands move point in the logical order, so the cursor will sometimes jump when point traverses reordered bidirectional text. Similarly, a highlighted region covering a contiguous range of character positions may look discontinuous if the region spans reordered text. This is normal and similar to the behavior of other programs that support bidirectional text.

# 20 Major and Minor Modes

A *mode* is a set of definitions that customize Emacs and can be turned on and off while you edit. There are two varieties of modes: *major modes*, which are mutually exclusive and used for editing particular kinds of text, and *minor modes*, which provide features that users can enable individually.

This chapter describes how to write both major and minor modes, how to indicate them in the mode line, and how they run hooks supplied by the user. For related topics such as keymaps and syntax tables, see Section 33.3.1 [Keymaps], page 703, and ⟨undefined⟩ [Syntax Tables], page ⟨undefined⟩.

## 20.1 Hooks

A *hook* is a variable where you can store a function or functions to be called on a particular occasion by an existing program. Emacs provides hooks for the sake of customization. Most often, hooks are set up in the init file (see Section 33.4 [Init File], page 711), but Lisp programs can set them also. See ⟨undefined⟩ [Standard Hooks], page ⟨undefined⟩, for a list of some standard hook variables.

Most of the hooks in Emacs are *normal hooks*. These variables contain lists of functions to be called with no arguments. By convention, whenever the hook name ends in '`-hook`', that tells you it is normal. We try to make all hooks normal, as much as possible, so that you can use them in a uniform way.

Every major mode command is supposed to run a normal hook called the *mode hook* as one of the last steps of initialization. This makes it easy for a user to customize the behavior of the mode, by overriding the buffer-local variable assignments already made by the mode. Most minor mode functions also run a mode hook at the end. But hooks are used in other contexts too. For example, the hook `suspend-hook` runs just before Emacs suspends itself (see ⟨undefined⟩ [Suspending Emacs], page ⟨undefined⟩).

The recommended way to add a hook function to a hook is by calling `add-hook` (see Section 20.1.2 [Setting Hooks], page 398). The hook functions may be any of the valid kinds of functions that `funcall` accepts (see ⟨undefined⟩ [What Is a Function], page ⟨undefined⟩). Most normal hook variables are initially void; `add-hook` knows how to deal with this. You can add hooks either globally or buffer-locally with `add-hook`.

If the hook variable's name does not end with '`-hook`', that indicates it is probably an *abnormal hook*. That means the hook functions are called with arguments, or their return values are used in some way. The hook's documentation says how the functions are called. You can use `add-hook` to add a function to an abnormal hook, but you must write the function to follow the hook's calling convention.

By convention, abnormal hook names end in '`-functions`'. If the variable's name ends in '`-function`', then its value is just a single function, not a list of functions.

### 20.1.1 Running Hooks

In this section, we document the `run-hooks` function, which is used to run a normal hook. We also document the functions for running various kinds of abnormal hooks.

`run-hooks` **&rest** *hookvars* [Function]

This function takes one or more normal hook variable names as arguments, and runs each hook in turn. Each argument should be a symbol that is a normal hook variable. These arguments are processed in the order specified.

If a hook variable has a non-`nil` value, that value should be a list of functions. `run-hooks` calls all the functions, one by one, with no arguments.

The hook variable's value can also be a single function—either a lambda expression or a symbol with a function definition—which `run-hooks` calls. But this usage is obsolete.

If the hook variable is buffer-local, the buffer-local variable will be used instead of the global variable. However, if the buffer-local variable contains the element `t`, the global hook variable will be run as well.

`run-hook-with-args` *hook* **&rest** *args* [Function]

This function runs an abnormal hook by calling all the hook functions in *hook*, passing each one the arguments *args*.

`run-hook-with-args-until-failure` *hook* **&rest** *args* [Function]

This function runs an abnormal hook by calling each hook function in turn, stopping if one of them "fails" by returning `nil`. Each hook function is passed the arguments *args*. If this function stops because one of the hook functions fails, it returns `nil`; otherwise it returns a non-`nil` value.

`run-hook-with-args-until-success` *hook* **&rest** *args* [Function]

This function runs an abnormal hook by calling each hook function, stopping if one of them "succeeds" by returning a non-`nil` value. Each hook function is passed the arguments *args*. If this function stops because one of the hook functions returns a non-`nil` value, it returns that value; otherwise it returns `nil`.

`with-wrapper-hook` *hook args* **&rest** *body* [Macro]

This macro runs the abnormal hook `hook` as a series of nested "wrapper functions" around the *body* forms. The effect is similar to nested `around` advices (see ⟨undefined⟩ [Around-Advice], page ⟨undefined⟩).

Each hook function should accept an argument list consisting of a function *fun*, followed by the additional arguments listed in *args*. The first hook function is passed a function *fun* that, if it is called with arguments *args*, performs *body* (i.e., the default operation). The *fun* passed to each successive hook function is constructed from all the preceding hook functions (and *body*); if this *fun* is called with arguments *args*, it does what the `with-wrapper-hook` call would if the preceding hook functions were the only ones in *hook*.

Each hook function may call its *fun* argument as many times as it wishes, including never. In that case, such a hook function acts to replace the default definition altogether, and any preceding hook functions. Of course, a subsequent hook function may do the same thing.

Each hook function definition is used to construct the *fun* passed to the next hook function in *hook*, if any. The last or "outermost" *fun* is called once to produce the overall effect.

When might you want to use a wrapper hook? The function `filter-buffer-substring` illustrates a common case. There is a basic functionality, performed by *body*—in this case, to extract a buffer-substring. Then any number of hook functions can act in sequence to modify that string, before returning the final result. A wrapper-hook also allows for a hook function to completely replace the default definition (by not calling *fun*).

`run-hook-wrapped` *hook wrap-function* **&rest** *args* [Function]
This function is similar to `run-hook-with-args-until-success`. Like that function, it runs the functions on the abnormal hook `hook`, stopping at the first one that returns non-`nil`. Instead of calling the hook functions directly, though, it actually calls `wrap-function` with arguments `fun` and `args`.

### 20.1.2 Setting Hooks

Here's an example that uses a mode hook to turn on Auto Fill mode when in Lisp Interaction mode:

```
(add-hook 'lisp-interaction-mode-hook 'auto-fill-mode)
```

`add-hook` *hook function* **&optional** *append local* [Function]
This function is the handy way to add function *function* to hook variable *hook*. You can use it for abnormal hooks as well as for normal hooks. *function* can be any Lisp function that can accept the proper number of arguments for *hook*. For example,

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

adds `my-text-hook-function` to the hook called `text-mode-hook`.

If *function* is already present in *hook* (comparing using `equal`), then `add-hook` does not add it a second time.

If *function* has a non-`nil` property `permanent-local-hook`, then `kill-all-local-variables` (or changing major modes) won't delete it from the hook variable's local value.

For a normal hook, hook functions should be designed so that the order in which they are executed does not matter. Any dependence on the order is asking for trouble. However, the order is predictable: normally, *function* goes at the front of the hook list, so it is executed first (barring another `add-hook` call). If the optional argument *append* is non-`nil`, the new hook function goes at the end of the hook list and is executed last.

`add-hook` can handle the cases where *hook* is void or its value is a single function; it sets or changes the value to a list of functions.

If *local* is non-`nil`, that says to add *function* to the buffer-local hook list instead of to the global hook list. This makes the hook buffer-local and adds `t` to the buffer-local value. The latter acts as a flag to run the hook functions in the default value as well as in the local value.

`remove-hook` *hook function* **&optional** *local* [Function]
This function removes *function* from the hook variable *hook*. It compares *function* with elements of *hook* using `equal`, so it works for both symbols and lambda expressions.

If *local* is non-`nil`, that says to remove *function* from the buffer-local hook list instead of from the global hook list.

## 20.2 Major Modes

Major modes specialize Emacs for editing particular kinds of text. Each buffer has one major mode at a time. Every major mode is associated with a *major mode command*, whose name should end in '`-mode`'. This command takes care of switching to that mode in the current buffer, by setting various buffer-local variables such as a local keymap. See Section 20.2.1 [Major Mode Conventions], page 399.

The least specialized major mode is called *Fundamental mode*, which has no mode-specific definitions or variable settings.

`fundamental-mode` [Command]

This is the major mode command for Fundamental mode. Unlike other mode commands, it does *not* run any mode hooks (see Section 20.2.1 [Major Mode Conventions], page 399), since you are not supposed to customize this mode.

The easiest way to write a major mode is to use the macro `define-derived-mode`, which sets up the new mode as a variant of an existing major mode. See Section 20.2.4 [Derived Modes], page 405. We recommend using `define-derived-mode` even if the new mode is not an obvious derivative of another mode, as it automatically enforces many coding conventions for you. See Section 20.2.5 [Basic Major Modes], page 407, for common modes to derive from.

The standard GNU Emacs Lisp directory tree contains the code for several major modes, in files such as '`text-mode.el`', '`texinfo.el`', '`lisp-mode.el`', and '`rmail.el`'. You can study these libraries to see how modes are written.

`major-mode` [User Option]

The buffer-local value of this variable holds the symbol for the current major mode. Its default value holds the default major mode for new buffers. The standard default value is `fundamental-mode`.

If the default value is `nil`, then whenever Emacs creates a new buffer via a command such as `C-x b` (`switch-to-buffer`), the new buffer is put in the major mode of the previously current buffer. As an exception, if the major mode of the previous buffer has a `mode-class` symbol property with value `special`, the new buffer is put in Fundamental mode (see Section 20.2.1 [Major Mode Conventions], page 399).

### 20.2.1 Major Mode Conventions

The code for every major mode should follow various coding conventions, including conventions for local keymap and syntax table initialization, function and variable names, and hooks.

If you use the `define-derived-mode` macro, it will take care of many of these conventions automatically. See Section 20.2.4 [Derived Modes], page 405. Note also that Fundamental mode is an exception to many of these conventions, because it represents the default state of Emacs.

The following list of conventions is only partial. Each major mode should aim for consistency in general with other Emacs major modes, as this makes Emacs as a whole more

coherent. It is impossible to list here all the possible points where this issue might come up; if the Emacs developers point out an area where your major mode deviates from the usual conventions, please make it compatible.

- Define a major mode command whose name ends in '`-mode`'. When called with no arguments, this command should switch to the new mode in the current buffer by setting up the keymap, syntax table, and buffer-local variables in an existing buffer. It should not change the buffer's contents.
- Write a documentation string for this command that describes the special commands available in this mode. See Section 20.2.3 [Mode Help], page 405.

  The documentation string may include the special documentation substrings, '`\[`*`command`*`]`', '`\{`*`keymap`*`}`', and '`\<`*`keymap`*`>`', which allow the help display to adapt automatically to the user's own key bindings. See Section 7.3 [Keys in Documentation], page 82.
- The major mode command should start by calling `kill-all-local-variables`. This runs the normal hook `change-major-mode-hook`, then gets rid of the buffer-local variables of the major mode previously in effect. See ⟨undefined⟩ [Creating Buffer-Local], page ⟨undefined⟩.
- The major mode command should set the variable `major-mode` to the major mode command symbol. This is how `describe-mode` discovers which documentation to print.
- The major mode command should set the variable `mode-name` to the "pretty" name of the mode, usually a string (but see Section 20.4.2 [Mode Line Data], page 419, for other possible forms). The name of the mode appears in the mode line.
- Since all global names are in the same name space, all the global variables, constants, and functions that are part of the mode should have names that start with the major mode name (or with an abbreviation of it if the name is long). See ⟨undefined⟩ [Coding Conventions], page ⟨undefined⟩.
- In a major mode for editing some kind of structured text, such as a programming language, indentation of text according to structure is probably useful. So the mode should set `indent-line-function` to a suitable function, and probably customize other variables for indentation. See Section 20.7 [Auto-Indentation], page 440.
- The major mode should usually have its own keymap, which is used as the local keymap in all buffers in that mode. The major mode command should call `use-local-map` to install this local map. See ⟨undefined⟩ [Active Keymaps], page ⟨undefined⟩, for more information.

  This keymap should be stored permanently in a global variable named *`modename`*`-mode-map`. Normally the library that defines the mode sets this variable.

  See ⟨undefined⟩ [Tips for Defining], page ⟨undefined⟩, for advice about how to write the code to set up the mode's keymap variable.
- The key sequences bound in a major mode keymap should usually start with `C-c`, followed by a control character, a digit, or `{`, `}`, `<`, `>`, `:` or `;`. The other punctuation characters are reserved for minor modes, and ordinary letters are reserved for users.

  A major mode can also rebind the keys `M-n`, `M-p` and `M-s`. The bindings for `M-n` and `M-p` should normally be some kind of "moving forward and backward", but this does not necessarily mean cursor motion.

It is legitimate for a major mode to rebind a standard key sequence if it provides a command that does "the same job" in a way better suited to the text this mode is used for. For example, a major mode for editing a programming language might redefine `C-M-a` to "move to the beginning of a function" in a way that works better for that language.

It is also legitimate for a major mode to rebind a standard key sequence whose standard meaning is rarely useful in that mode. For instance, minibuffer modes rebind `M-r`, whose standard meaning is rarely of any use in the minibuffer. Major modes such as Dired or Rmail that do not allow self-insertion of text can reasonably redefine letters and other printing characters as special commands.

- Major modes for editing text should not define RET to do anything other than insert a newline. However, it is ok for specialized modes for text that users don't directly edit, such as Dired and Info modes, to redefine RET to do something entirely different.
- Major modes should not alter options that are primarily a matter of user preference, such as whether Auto-Fill mode is enabled. Leave this to each user to decide. However, a major mode should customize other variables so that Auto-Fill mode will work usefully *if* the user decides to use it.
- The mode may have its own syntax table or may share one with other related modes. If it has its own syntax table, it should store this in a variable named `modename-mode-syntax-table`. See ⟨undefined⟩ [Syntax Tables], page ⟨undefined⟩.
- If the mode handles a language that has a syntax for comments, it should set the variables that define the comment syntax. See Section "Options Controlling Comments" in *The GNU Emacs Manual*.
- The mode may have its own abbrev table or may share one with other related modes. If it has its own abbrev table, it should store this in a variable named `modename-mode-abbrev-table`. If the major mode command defines any abbrevs itself, it should pass `t` for the *system-flag* argument to `define-abbrev`. See Section 26.2 [Defining Abbrevs], page 582.
- The mode should specify how to do highlighting for Font Lock mode, by setting up a buffer-local value for the variable `font-lock-defaults` (see Section 20.6 [Font Lock Mode], page 429).
- Each face that the mode defines should, if possible, inherit from an existing Emacs face. See Section 11.12.8 [Basic Faces], page 149, and Section 20.6.7 [Faces for Font Lock], page 436.
- The mode should specify how Imenu should find the definitions or sections of a buffer, by setting up a buffer-local value for the variable `imenu-generic-expression`, for the two variables `imenu-prev-index-position-function` and `imenu-extract-index-name-function`, or for the variable `imenu-create-index-function` (see Section 23.2.3 [Imenu], page 516).
- The mode can specify a local value for `eldoc-documentation-function` to tell ElDoc mode how to handle this mode.
- The mode can specify how to complete various keywords by adding one or more buffer-local entries to the special hook `completion-at-point-functions`. See ⟨undefined⟩ [Completion in Buffers], page ⟨undefined⟩.

- To make a buffer-local binding for an Emacs customization variable, use `make-local-variable` in the major mode command, not `make-variable-buffer-local`. The latter function would make the variable local to every buffer in which it is subsequently set, which would affect buffers that do not use this mode. It is undesirable for a mode to have such global effects. See ⟨undefined⟩ [Buffer-Local Variables], page ⟨undefined⟩.

  With rare exceptions, the only reasonable way to use `make-variable-buffer-local` in a Lisp package is for a variable which is used only within that package. Using it on a variable used by other packages would interfere with them.
- Each major mode should have a normal *mode hook* named *`modename`*`-mode-hook`. The very last thing the major mode command should do is to call `run-mode-hooks`. This runs the normal hook `change-major-mode-after-body-hook`, the mode hook, and then the normal hook `after-change-major-mode-hook`. See Section 20.2.6 [Mode Hooks], page 408.
- The major mode command may start by calling some other major mode command (called the *parent mode*) and then alter some of its settings. A mode that does this is called a *derived mode*. The recommended way to define one is to use the `define-derived-mode` macro, but this is not required. Such a mode should call the parent mode command inside a `delay-mode-hooks` form. (Using `define-derived-mode` does this automatically.) See Section 20.2.4 [Derived Modes], page 405, and Section 20.2.6 [Mode Hooks], page 408.
- If something special should be done if the user switches a buffer from this mode to any other major mode, this mode can set up a buffer-local value for `change-major-mode-hook` (see ⟨undefined⟩ [Creating Buffer-Local], page ⟨undefined⟩).
- If this mode is appropriate only for specially-prepared text produced by the mode itself (rather than by the user typing at the keyboard or by an external file), then the major mode command symbol should have a property named `mode-class` with value `special`, put on as follows:

  ```
  (put 'funny-mode 'mode-class 'special)
  ```

  This tells Emacs that new buffers created while the current buffer is in Funny mode should not be put in Funny mode, even though the default value of `major-mode` is `nil`. By default, the value of `nil` for `major-mode` means to use the current buffer's major mode when creating new buffers (see Section 20.2.2 [Auto Major Mode], page 403), but with such `special` modes, Fundamental mode is used instead. Modes such as Dired, Rmail, and Buffer List use this feature.

  The function `view-buffer` does not enable View mode in buffers whose mode-class is special, because such modes usually provide their own View-like bindings.

  The `define-derived-mode` macro automatically marks the derived mode as special if the parent mode is special. Special mode is a convenient parent for such modes to inherit from; See Section 20.2.5 [Basic Major Modes], page 407.
- If you want to make the new mode the default for files with certain recognizable names, add an element to `auto-mode-alist` to select the mode for those file names (see Section 20.2.2 [Auto Major Mode], page 403). If you define the mode command to autoload, you should add this element in the same file that calls `autoload`. If you use an autoload cookie for the mode command, you can also use an autoload cookie for the form that adds the element (see ⟨undefined⟩ [autoload cookie], page ⟨undefined⟩). If

you do not autoload the mode command, it is sufficient to add the element in the file that contains the mode definition.

- The top-level forms in the file defining the mode should be written so that they may be evaluated more than once without adverse consequences. For instance, use `defvar` or `defcustom` to set mode-related variables, so that they are not reinitialized if they already have a value (see ⟨undefined⟩ [Defining Variables], page ⟨undefined⟩).

### 20.2.2 How Emacs Chooses a Major Mode

When Emacs visits a file, it automatically selects a major mode for the buffer based on information in the file name or in the file itself. It also processes local variables specified in the file text.

`normal-mode` **&optional** *find-file* [Command]

This function establishes the proper major mode and buffer-local variable bindings for the current buffer. First it calls `set-auto-mode` (see below), then it runs `hack-local-variables` to parse, and bind or evaluate as appropriate, the file's local variables (see ⟨undefined⟩ [File Local Variables], page ⟨undefined⟩).

If the *find-file* argument to `normal-mode` is non-`nil`, `normal-mode` assumes that the `find-file` function is calling it. In this case, it may process local variables in the '`-*-`' line or at the end of the file. The variable `enable-local-variables` controls whether to do so. See Section "Local Variables in Files" in *The GNU Emacs Manual*, for the syntax of the local variables section of a file.

If you run `normal-mode` interactively, the argument *find-file* is normally `nil`. In this case, `normal-mode` unconditionally processes any file local variables.

The function calls `set-auto-mode` to choose a major mode. If this does not specify a mode, the buffer stays in the major mode determined by the default value of `major-mode` (see below).

`normal-mode` uses `condition-case` around the call to the major mode command, so errors are caught and reported as a '`File mode specification error`', followed by the original error message.

`set-auto-mode` **&optional** *keep-mode-if-same* [Function]

This function selects the major mode that is appropriate for the current buffer. It bases its decision (in order of precedence) on the '`-*-`' line, on any '`mode:`' local variable near the end of a file, on the '`#!`' line (using `interpreter-mode-alist`), on the text at the beginning of the buffer (using `magic-mode-alist`), and finally on the visited file name (using `auto-mode-alist`). See Section "How Major Modes are Chosen" in *The GNU Emacs Manual*. If `enable-local-variables` is `nil`, `set-auto-mode` does not check the '`-*-`' line, or near the end of the file, for any mode tag.

There are some file types where it is not appropriate to scan the file contents for a mode specifier. For example, a tar archive may happen to contain, near the end of the file, a member file that has a local variables section specifying a mode for that particular file. This should not be applied to the containing tar file. Similarly, a tiff image file might just happen to contain a first line that seems to match the '`-*-`' pattern. For these reasons, both these file extensions are members of the list

`inhibit-local-variables-regexps`. Add patterns to this list to prevent Emacs searching them for local variables of any kind (not just mode specifiers).

If *keep-mode-if-same* is non-`nil`, this function does not call the mode command if the buffer is already in the proper major mode. For instance, `set-visited-file-name` sets this to `t` to avoid killing buffer local variables that the user may have set.

`set-buffer-major-mode` *buffer* [Function]

This function sets the major mode of *buffer* to the default value of `major-mode`; if that is `nil`, it uses the current buffer's major mode (if that is suitable). As an exception, if *buffer*'s name is '`*scratch*`', it sets the mode to `initial-major-mode`.

The low-level primitives for creating buffers do not use this function, but medium-level commands such as `switch-to-buffer` and `find-file-noselect` use it whenever they create buffers.

`initial-major-mode` [User Option]

The value of this variable determines the major mode of the initial '`*scratch*`' buffer. The value should be a symbol that is a major mode command. The default value is `lisp-interaction-mode`.

`interpreter-mode-alist` [Variable]

This variable specifies major modes to use for scripts that specify a command interpreter in a '`#!`' line. Its value is an alist with elements of the form `(`*`interpreter`* `.` *`mode`*`)`; for example, `("perl" . perl-mode)` is one element present by default. The element says to use mode *mode* if the file specifies an interpreter which matches *interpreter*.

`magic-mode-alist` [Variable]

This variable's value is an alist with elements of the form `(`*`regexp`* `.` *`function`*`)`, where *regexp* is a regular expression and *function* is a function or `nil`. After visiting a file, `set-auto-mode` calls *function* if the text at the beginning of the buffer matches *regexp* and *function* is non-`nil`; if *function* is `nil`, `auto-mode-alist` gets to decide the mode.

`magic-fallback-mode-alist` [Variable]

This works like `magic-mode-alist`, except that it is handled only if `auto-mode-alist` does not specify a mode for this file.

`auto-mode-alist` [Variable]

This variable contains an association list of file name patterns (regular expressions) and corresponding major mode commands. Usually, the file name patterns test for suffixes, such as '`.el`' and '`.c`', but this need not be the case. An ordinary element of the alist looks like `(`*`regexp`* `.` *`mode-function`*`)`.

For example,

```
(("\\`/tmp/fol/" . text-mode)
 ("\\.texinfo\\'" . texinfo-mode)
 ("\\.texi\\'" . texinfo-mode)
 ("\\.el\\'" . emacs-lisp-mode)
 ("\\.c\\'" . c-mode)
 ("\\.h\\'" . c-mode)
 ...)
```

When you visit a file whose expanded file name (see Section 15.8.4 [File Name Expansion], page 255), with version numbers and backup suffixes removed using `file-name-sans-versions` (see Section 15.8.1 [File Name Components], page 251), matches a *regexp*, `set-auto-mode` calls the corresponding *mode-function*. This feature enables Emacs to select the proper major mode for most files.

If an element of `auto-mode-alist` has the form `(regexp function t)`, then after calling *function*, Emacs searches `auto-mode-alist` again for a match against the portion of the file name that did not match before. This feature is useful for uncompression packages: an entry of the form `("\\.gz\\'" function t)` can uncompress the file and then put the uncompressed file in the proper mode according to the name sans '`.gz`'.

Here is an example of how to prepend several pattern pairs to `auto-mode-alist`. (You might use this sort of expression in your init file.)

```
(setq auto-mode-alist
  (append
   ;; File name (within directory) starts with a dot.
   '(("/\\.[^/]*\\'" . fundamental-mode)
     ;; File name has no dot.
     ("/[^\\./]*\\'" . fundamental-mode)
     ;; File name ends in '.C'.
     ("\\.C\\'" . c++-mode))
   auto-mode-alist))
```

### 20.2.3 Getting Help about a Major Mode

The `describe-mode` function provides information about major modes. It is normally bound to `C-h m`. It uses the value of the variable `major-mode` (see Section 20.2 [Major Modes], page 399), which is why every major mode command needs to set that variable.

`describe-mode` **&optional** *buffer* [Command]

This command displays the documentation of the current buffer's major mode and minor modes. It uses the `documentation` function to retrieve the documentation strings of the major and minor mode commands (see Section 7.2 [Accessing Documentation], page 80).

If called from Lisp with a non-nil *buffer* argument, this function displays the documentation for that buffer's major and minor modes, rather than those of the current buffer.

### 20.2.4 Defining Derived Modes

The recommended way to define a new major mode is to derive it from an existing one using `define-derived-mode`. If there is no closely related mode, you should inherit from either `text-mode`, `special-mode`, or `prog-mode`. See Section 20.2.5 [Basic Major Modes], page 407. If none of these are suitable, you can inherit from `fundamental-mode` (see Section 20.2 [Major Modes], page 399).

`define-derived-mode` *variant parent name docstring keyword-args. . . body. . .* [Macro]

This macro defines *variant* as a major mode command, using *name* as the string form of the mode name. *variant* and *parent* should be unquoted symbols.

The new command *variant* is defined to call the function *parent*, then override certain aspects of that parent mode:

- The new mode has its own sparse keymap, named `variant-map`. `define-derived-mode` makes the parent mode's keymap the parent of the new map, unless `variant-map` is already set and already has a parent.
- The new mode has its own syntax table, kept in the variable `variant-syntax-table`, unless you override this using the `:syntax-table` keyword (see below). `define-derived-mode` makes the parent mode's syntax-table the parent of `variant-syntax-table`, unless the latter is already set and already has a parent different from the standard syntax table.
- The new mode has its own abbrev table, kept in the variable `variant-abbrev-table`, unless you override this using the `:abbrev-table` keyword (see below).
- The new mode has its own mode hook, `variant-hook`. It runs this hook, after running the hooks of its ancestor modes, with `run-mode-hooks`, as the last thing it does. See Section 20.2.6 [Mode Hooks], page 408.

In addition, you can specify how to override other aspects of *parent* with *body*. The command *variant* evaluates the forms in *body* after setting up all its usual overrides, just before running the mode hooks.

If *parent* has a non-`nil` `mode-class` symbol property, then `define-derived-mode` sets the `mode-class` property of *variant* to the same value. This ensures, for example, that if *parent* is a special mode, then *variant* is also a special mode (see Section 20.2.1 [Major Mode Conventions], page 399).

You can also specify `nil` for *parent*. This gives the new mode no parent. Then `define-derived-mode` behaves as described above, but, of course, omits all actions connected with *parent*.

The argument *docstring* specifies the documentation string for the new mode. `define-derived-mode` adds some general information about the mode's hook, followed by the mode's keymap, at the end of this documentation string. If you omit *docstring*, `define-derived-mode` generates a documentation string.

The *keyword-args* are pairs of keywords and values. The values are evaluated. The following keywords are currently supported:

`:syntax-table`
: You can use this to explicitly specify a syntax table for the new mode. If you specify a `nil` value, the new mode uses the same syntax table as *parent*, or the standard syntax table if *parent* is `nil`. (Note that this does *not* follow the convention used for non-keyword arguments that a `nil` value is equivalent with not specifying the argument.)

`:abbrev-table`
: You can use this to explicitly specify an abbrev table for the new mode. If you specify a `nil` value, the new mode uses the same abbrev table as *parent*, or `fundamental-mode-abbrev-table` if *parent* is `nil`. (Again, a `nil` value is *not* equivalent to not specifying this keyword.)

`:group`
: If this is specified, the value should be the customization group for this mode. (Not all major modes have one.) Only the (still experimental and

unadvertised) command `customize-mode` currently uses this. `define-derived-mode` does *not* automatically define the specified customization group.

Here is a hypothetical example:

```
(define-derived-mode hypertext-mode
  text-mode "Hypertext"
  "Major mode for hypertext.
\\{hypertext-mode-map}"
  (setq case-fold-search nil))

(define-key hypertext-mode-map
  [down-mouse-3] 'do-hyper-link)
```

Do not write an `interactive` spec in the definition; `define-derived-mode` does that automatically.

`derived-mode-p` **&rest** *modes* [Function]
This function returns non-`nil` if the current major mode is derived from any of the major modes given by the symbols *modes*.

### 20.2.5 Basic Major Modes

Apart from Fundamental mode, there are three major modes that other major modes commonly derive from: Text mode, Prog mode, and Special mode. While Text mode is useful in its own right (e.g., for editing files ending in '`.txt`'), Prog mode and Special mode exist mainly to let other modes derive from them.

As far as possible, new major modes should be derived, either directly or indirectly, from one of these three modes. One reason is that this allows users to customize a single mode hook (e.g., `prog-mode-hook`) for an entire family of relevant modes (e.g., all programming language modes).

`text-mode` [Command]
Text mode is a major mode for editing human languages. It defines the '`"`' and '`\`' characters as having punctuation syntax (see ⟨undefined⟩ [Syntax Class Table], page ⟨undefined⟩), and binds `M-TAB` to `ispell-complete-word` (see Section "Spelling" in *The GNU Emacs Manual*).

An example of a major mode derived from Text mode is HTML mode. See Section "SGML and HTML Modes" in *The GNU Emacs Manual*.

`prog-mode` [Command]
Prog mode is a basic major mode for buffers containing programming language source code. Most of the programming language major modes built into Emacs are derived from it.

Prog mode binds `parse-sexp-ignore-comments` to `t` (see ⟨undefined⟩ [Motion via Parsing], page ⟨undefined⟩) and `bidi-paragraph-direction` to `left-to-right` (see Section 11.23 [Bidirectional Display], page 195).

`special-mode` [Command]

Special mode is a basic major mode for buffers containing text that is produced specially by Emacs, rather than directly from a file. Major modes derived from Special mode are given a `mode-class` property of `special` (see Section 20.2.1 [Major Mode Conventions], page 399).

Special mode sets the buffer to read-only. Its keymap defines several common bindings, including `q` for `quit-window` and `g` for `revert-buffer` (see ⟨undefined⟩ [Reverting], page ⟨undefined⟩).

An example of a major mode derived from Special mode is Buffer Menu mode, which is used by the '`*Buffer List*`' buffer. See Section "Listing Existing Buffers" in *The GNU Emacs Manual*.

In addition, modes for buffers of tabulated data can inherit from Tabulated List mode, which is in turn derived from Special mode. See Section 20.2.7 [Tabulated List Mode], page 409.

### 20.2.6 Mode Hooks

Every major mode command should finish by running the mode-independent normal hook `change-major-mode-after-body-hook`, its mode hook, and the normal hook `after-change-major-mode-hook`. It does this by calling `run-mode-hooks`. If the major mode is a derived mode, that is if it calls another major mode (the parent mode) in its body, it should do this inside `delay-mode-hooks` so that the parent won't run these hooks itself. Instead, the derived mode's call to `run-mode-hooks` runs the parent's mode hook too. See Section 20.2.1 [Major Mode Conventions], page 399.

Emacs versions before Emacs 22 did not have `delay-mode-hooks`. Versions before 24 did not have `change-major-mode-after-body-hook`. When user-implemented major modes do not use `run-mode-hooks` and have not been updated to use these newer features, they won't entirely follow these conventions: they may run the parent's mode hook too early, or fail to run `after-change-major-mode-hook`. If you encounter such a major mode, please correct it to follow these conventions.

When you defined a major mode using `define-derived-mode`, it automatically makes sure these conventions are followed. If you define a major mode "by hand", not using `define-derived-mode`, use the following functions to handle these conventions automatically.

`run-mode-hooks` **&rest** *hookvars* [Function]

Major modes should run their mode hook using this function. It is similar to `run-hooks` (see Section 33.2.2 [Hooks], page 696), but it also runs `change-major-mode-after-body-hook` and `after-change-major-mode-hook`.

When this function is called during the execution of a `delay-mode-hooks` form, it does not run the hooks immediately. Instead, it arranges for the next call to `run-mode-hooks` to run them.

`delay-mode-hooks` *body...* [Macro]

When one major mode command calls another, it should do so inside of `delay-mode-hooks`.

This macro executes *body*, but tells all `run-mode-hooks` calls during the execution of *body* to delay running their hooks. The hooks will actually run during the next call to `run-mode-hooks` after the end of the `delay-mode-hooks` construct.

`change-major-mode-after-body-hook` [Variable]
This is a normal hook run by `run-mode-hooks`. It is run before the mode hooks.

`after-change-major-mode-hook` [Variable]
This is a normal hook run by `run-mode-hooks`. It is run at the very end of every properly-written major mode command.

### 20.2.7 Tabulated List mode

Tabulated List mode is a major mode for displaying tabulated data, i.e., data consisting of *entries*, each entry occupying one row of text with its contents divided into columns. Tabulated List mode provides facilities for pretty-printing rows and columns, and sorting the rows according to the values in each column. It is derived from Special mode (see Section 20.2.5 [Basic Major Modes], page 407).

Tabulated List mode is intended to be used as a parent mode by a more specialized major mode. Examples include Process Menu mode (see ⟨undefined⟩ [Process Information], page ⟨undefined⟩) and Package Menu mode (see Section "Package Menu" in *The GNU Emacs Manual*).

Such a derived mode should use `define-derived-mode` in the usual way, specifying `tabulated-list-mode` as the second argument (see Section 20.2.4 [Derived Modes], page 405). The body of the `define-derived-mode` form should specify the format of the tabulated data, by assigning values to the variables documented below; then, it should call the function `tabulated-list-init-header` to initialize the header line.

The derived mode should also define a *listing command*. This, not the mode command, is what the user calls (e.g., `M-x list-processes`). The listing command should create or switch to a buffer, turn on the derived mode, specify the tabulated data, and finally call `tabulated-list-print` to populate the buffer.

`tabulated-list-format` [Variable]
This buffer-local variable specifies the format of the Tabulated List data. Its value should be a vector. Each element of the vector represents a data column, and should be a list `(`*`name width sort`*`)`, where

- *name* is the column's name (a string).
- *width* is the width to reserve for the column (an integer). This is meaningless for the last column, which runs to the end of each line.
- *sort* specifies how to sort entries by the column. If `nil`, the column cannot be used for sorting. If `t`, the column is sorted by comparing string values. Otherwise, this should be a predicate function for `sort` (see ⟨undefined⟩ [Rearrangement], page ⟨undefined⟩), which accepts two arguments with the same form as the elements of `tabulated-list-entries` (see below).

`tabulated-list-entries` [Variable]
This buffer-local variable specifies the entries displayed in the Tabulated List buffer. Its value should be either a list, or a function.

If the value is a list, each list element corresponds to one entry, and should have the form `(id contents)`, where

- *id* is either `nil`, or a Lisp object that identifies the entry. If the latter, the cursor stays on the "same" entry when re-sorting entries. Comparison is done with `equal`.
- *contents* is a vector with the same number of elements as `tabulated-list-format`. Each vector element is either a string, which is inserted into the buffer as-is, or a list `(label . properties)`, which means to insert a text button by calling `insert-text-button` with *label* and *properties* as arguments (see Section 11.17.3 [Making Buttons], page 181).

  There should be no newlines in any of these strings.

Otherwise, the value should be a function which returns a list of the above form when called with no arguments.

`tabulated-list-revert-hook` [Variable]
This normal hook is run prior to reverting a Tabulated List buffer. A derived mode can add a function to this hook to recompute `tabulated-list-entries`.

`tabulated-list-printer` [Variable]
The value of this variable is the function called to insert an entry at point, including its terminating newline. The function should accept two arguments, *id* and *contents*, having the same meanings as in `tabulated-list-entries`. The default value is a function which inserts an entry in a straightforward way; a mode which uses Tabulated List mode in a more complex way can specify another function.

`tabulated-list-sort-key` [Variable]
The value of this variable specifies the current sort key for the Tabulated List buffer. If it is `nil`, no sorting is done. Otherwise, it should have the form `(name . flip)`, where *name* is a string matching one of the column names in `tabulated-list-format`, and *flip*, if non-`nil`, means to invert the sort order.

`tabulated-list-init-header` [Function]
This function computes and sets `header-line-format` for the Tabulated List buffer (see Section 20.4.7 [Header Lines], page 426), and assigns a keymap to the header line to allow sort entries by clicking on column headers.

Modes derived from Tabulated List mode should call this after setting the above variables (in particular, only after setting `tabulated-list-format`).

`tabulated-list-print` **&optional** *remember-pos* [Function]
This function populates the current buffer with entries. It should be called by the listing command. It erases the buffer, sorts the entries specified by `tabulated-list-entries` according to `tabulated-list-sort-key`, then calls the function specified by `tabulated-list-printer` to insert each entry.

If the optional argument *remember-pos* is non-`nil`, this function looks for the *id* element on the current line, if any, and tries to move to that entry after all the entries are (re)inserted.

### 20.2.8 Generic Modes

*Generic modes* are simple major modes with basic support for comment syntax and Font Lock mode. To define a generic mode, use the macro `define-generic-mode`. See the file '`generic-x.el`' for some examples of the use of `define-generic-mode`.

`define-generic-mode` *mode comment-list keyword-list font-lock-list auto-mode-list function-list* **&optional** *docstring* [Macro]

This macro defines a generic mode command named *mode* (a symbol, not quoted). The optional argument *docstring* is the documentation for the mode command. If you do not supply it, `define-generic-mode` generates one by default.

The argument *comment-list* is a list in which each element is either a character, a string of one or two characters, or a cons cell. A character or a string is set up in the mode's syntax table as a "comment starter". If the entry is a cons cell, the CAR is set up as a "comment starter" and the CDR as a "comment ender". (Use `nil` for the latter if you want comments to end at the end of the line.) Note that the syntax table mechanism has limitations about what comment starters and enders are actually possible. See ⟨undefined⟩ [Syntax Tables], page ⟨undefined⟩.

The argument *keyword-list* is a list of keywords to highlight with `font-lock-keyword-face`. Each keyword should be a string. Meanwhile, *font-lock-list* is a list of additional expressions to highlight. Each element of this list should have the same form as an element of `font-lock-keywords`. See Section 20.6.2 [Search-based Fontification], page 430.

The argument *auto-mode-list* is a list of regular expressions to add to the variable `auto-mode-alist`. They are added by the execution of the `define-generic-mode` form, not by expanding the macro call.

Finally, *function-list* is a list of functions for the mode command to call for additional setup. It calls these functions just before it runs the mode hook variable *`mode`*`-hook`.

### 20.2.9 Major Mode Examples

Text mode is perhaps the simplest mode besides Fundamental mode. Here are excerpts from '`text-mode.el`' that illustrate many of the conventions listed above:

```
;; Create the syntax table for this mode.
(defvar text-mode-syntax-table
  (let ((st (make-syntax-table)))
    (modify-syntax-entry ?\" ".   " st)
    (modify-syntax-entry ?\\ ".   " st)
    ;; Add ‘p’ so M-c on ‘hello’ leads to ‘Hello’, not ‘hello’.
    (modify-syntax-entry ?’ "w p" st)
    st)
  "Syntax table used while in ‘text-mode’.")

;; Create the keymap for this mode.
(defvar text-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\e\t" ’ispell-complete-word)
    map)
  "Keymap for ‘text-mode’.
Many other modes, such as ‘mail-mode’, ‘outline-mode’ and
‘indented-text-mode’, inherit all the commands defined in this map.")
```

Here is how the actual mode command is defined now:

```
(define-derived-mode text-mode nil "Text"
  "Major mode for editing text written for humans to read.
In this mode, paragraphs are delimited only by blank or white lines.
You can thus get the full benefit of adaptive filling
 (see the variable ‘adaptive-fill-mode’).
\\{text-mode-map}
Turning on Text mode runs the normal hook ‘text-mode-hook’."
  (set (make-local-variable ’text-mode-variant) t)
  (set (make-local-variable ’require-final-newline)
       mode-require-final-newline)
  (set (make-local-variable ’indent-line-function) ’indent-relative))
```

(The last line is redundant nowadays, since `indent-relative` is the default value, and we'll delete it in a future version.)

The three Lisp modes (Lisp mode, Emacs Lisp mode, and Lisp Interaction mode) have more features than Text mode and the code is correspondingly more complicated. Here are excerpts from '`lisp-mode.el`' that illustrate how these modes are written.

Here is how the Lisp mode syntax and abbrev tables are defined:

```
;; Create mode-specific table variables.
(defvar lisp-mode-abbrev-table nil)
(define-abbrev-table ’lisp-mode-abbrev-table ())

(defvar lisp-mode-syntax-table
  (let ((table (copy-syntax-table emacs-lisp-mode-syntax-table)))
    (modify-syntax-entry ?\[ "_   " table)
    (modify-syntax-entry ?\] "_   " table)
    (modify-syntax-entry ?# "’ 14" table)
    (modify-syntax-entry ?| "\" 23bn" table)
    table)
  "Syntax table used in ‘lisp-mode’.")
```

The three modes for Lisp share much of their code. For instance, each calls the following function to set various variables:

```
(defun lisp-mode-variables (&optional syntax keywords-case-insensitive)
  (when syntax
    (set-syntax-table lisp-mode-syntax-table))
  (setq local-abbrev-table lisp-mode-abbrev-table)
  ...
```

Amongst other things, this function sets up the `comment-start` variable to handle Lisp comments:

```
  (make-local-variable ’comment-start)
  (setq comment-start ";")
  ...
```

Each of the different Lisp modes has a slightly different keymap. For example, Lisp mode binds `C-c C-z` to `run-lisp`, but the other Lisp modes do not. However, all Lisp modes have some commands in common. The following code sets up the common commands:

```
(defvar lisp-mode-shared-map
  (let ((map (make-sparse-keymap)))
    (define-key map "\e\C-q" ’indent-sexp)
    (define-key map "\177" ’backward-delete-char-untabify)
    map)
  "Keymap for commands shared by all sorts of Lisp modes.")
```

And here is the code to set up the keymap for Lisp mode:

```
(defvar lisp-mode-map
  (let ((map (make-sparse-keymap))
(menu-map (make-sparse-keymap "Lisp")))
    (set-keymap-parent map lisp-mode-shared-map)
    (define-key map "\e\C-x" 'lisp-eval-defun)
    (define-key map "\C-c\C-z" 'run-lisp)
    ...
    map)
  "Keymap for ordinary Lisp mode.
All commands in `lisp-mode-shared-map' are inherited by this map.")
```

Finally, here is the major mode command for Lisp mode:

```
(define-derived-mode lisp-mode prog-mode "Lisp"
  "Major mode for editing Lisp code for Lisps other than GNU Emacs Lisp.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.

\\{lisp-mode-map}
Note that `run-lisp' may be used either to start an inferior Lisp job
or to switch back to an existing one.

Entry to this mode calls the value of `lisp-mode-hook'
if that value is non-nil."
  (lisp-mode-variables nil t)
  (set (make-local-variable 'find-tag-default-function)
       'lisp-find-tag-default)
  (set (make-local-variable 'comment-start-skip)
       "\\(\\(^\\|[^\\\\\n]\\)\\(\\\\\\\\)*\\)\\(;+\\|#|\\) *")
  (setq imenu-case-fold-search t))
```

## 20.3 Minor Modes

A *minor mode* provides optional features that users may enable or disable independently of the choice of major mode. Minor modes can be enabled individually or in combination.

Most minor modes implement features that are independent of the major mode, and can thus be used with most major modes. For example, Auto Fill mode works with any major mode that permits text insertion. A few minor modes, however, are specific to a particular major mode. For example, Diff Auto Refine mode is a minor mode that is intended to be used only with Diff mode.

Ideally, a minor mode should have its desired effect regardless of the other minor modes in effect. It should be possible to activate and deactivate minor modes in any order.

`minor-mode-list` [Variable]

The value of this variable is a list of all minor mode commands.

### 20.3.1 Conventions for Writing Minor Modes

There are conventions for writing minor modes just as there are for major modes. These conventions are described below. The easiest way to follow them is to use the macro `define-minor-mode`. See Section 20.3.3 [Defining Minor Modes], page 415.

- Define a variable whose name ends in '`-mode`'. We call this the *mode variable*. The minor mode command should set this variable. The value will be `nil` if the mode is

disabled, and non-`nil` if the mode is enabled. The variable should be buffer-local if the minor mode is buffer-local.

This variable is used in conjunction with the `minor-mode-alist` to display the minor mode name in the mode line. It also determines whether the minor mode keymap is active, via `minor-mode-map-alist` (see ⟨undefined⟩ [Controlling Active Maps], page ⟨undefined⟩). Individual commands or hooks can also check its value.

- Define a command, called the *mode command*, whose name is the same as the mode variable. Its job is to set the value of the mode variable, plus anything else that needs to be done to actually enable or disable the mode's features.

  The mode command should accept one optional argument. If called interactively with no prefix argument, it should toggle the mode (i.e., enable if it is disabled, and disable if it is enabled). If called interactively with a prefix argument, it should enable the mode if the argument is positive and disable it otherwise.

  If the mode command is called from Lisp (i.e., non-interactively), it should enable the mode if the argument is omitted or `nil`; it should toggle the mode if the argument is the symbol `toggle`; otherwise it should treat the argument in the same way as for an interactive call with a numeric prefix argument, as described above.

  The following example shows how to implement this behavior (it is similar to the code generated by the `define-minor-mode` macro):

```
(interactive (list (or current-prefix-arg 'toggle)))
(let ((enable (if (eq arg 'toggle)
                  (not foo-mode) ; this mode's mode variable
                (> (prefix-numeric-value arg) 0))))
  (if enable
      do-enable
    do-disable))
```

  The reason for this somewhat complex behavior is that it lets users easily toggle the minor mode interactively, and also lets the minor mode be easily enabled in a mode hook, like this:

```
(add-hook 'text-mode-hook 'foo-mode)
```

  This behaves correctly whether or not `foo-mode` was already enabled, since the `foo-mode` mode command unconditionally enables the minor mode when it is called from Lisp with no argument. Disabling a minor mode in a mode hook is a little uglier:

```
(add-hook 'text-mode-hook (lambda () (foo-mode -1)))
```

  However, this is not very commonly done.

- Add an element to `minor-mode-alist` for each minor mode (see [Definition of minor-mode-alist], page 423), if you want to indicate the minor mode in the mode line. This element should be a list of the following form:

```
(mode-variable string)
```

  Here *mode-variable* is the variable that controls enabling of the minor mode, and *string* is a short string, starting with a space, to represent the mode in the mode line. These strings must be short so that there is room for several of them at once.

  When you add an element to `minor-mode-alist`, use `assq` to check for an existing element, to avoid duplication. For example:

```
(unless (assq 'leif-mode minor-mode-alist)
  (push '(leif-mode " Leif") minor-mode-alist))
```

or like this, using `add-to-list` (see ⟨undefined⟩ [List Variables], page ⟨undefined⟩):

```
(add-to-list 'minor-mode-alist '(leif-mode " Leif"))
```

In addition, several major mode conventions apply to minor modes as well: those regarding the names of global symbols, the use of a hook at the end of the initialization function, and the use of keymaps and other tables.

The minor mode should, if possible, support enabling and disabling via Custom (see Chapter 33 [Customization], page 686). To do this, the mode variable should be defined with `defcustom`, usually with `:type 'boolean`. If just setting the variable is not sufficient to enable the mode, you should also specify a `:set` method which enables the mode by invoking the mode command. Note in the variable's documentation string that setting the variable other than via Custom may not take effect. Also, mark the definition with an autoload cookie (see ⟨undefined⟩ [autoload cookie], page ⟨undefined⟩), and specify a `:require` so that customizing the variable will load the library that defines the mode. For example:

```
;;;###autoload
(defcustom msb-mode nil
  "Toggle msb-mode.
Setting this variable directly does not take effect;
use either \\[customize] or the function `msb-mode'."
  :set 'custom-set-minor-mode
  :initialize 'custom-initialize-default
  :version "20.4"
  :type    'boolean
  :group   'msb
  :require 'msb)
```

### 20.3.2 Keymaps and Minor Modes

Each minor mode can have its own keymap, which is active when the mode is enabled. To set up a keymap for a minor mode, add an element to the alist `minor-mode-map-alist`. See ⟨undefined⟩ [Definition of minor-mode-map-alist], page ⟨undefined⟩.

One use of minor mode keymaps is to modify the behavior of certain self-inserting characters so that they do something else as well as self-insert. (Another way to customize `self-insert-command` is through `post-self-insert-hook`. Apart from this, the facilities for customizing `self-insert-command` are limited to special cases, designed for abbrevs and Auto Fill mode. Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.)

The key sequences bound in a minor mode should consist of `C-c` followed by one of `.,/¿'"[]\|~!#$%^&*()-_+=`. (The other punctuation characters are reserved for major modes.)

### 20.3.3 Defining Minor Modes

The macro `define-minor-mode` offers a convenient way of implementing a mode in one self-contained definition.

`define-minor-mode` *mode doc* [*init-value* [*lighter* [*keymap*]]] [Macro]
*keyword-args... body...*

This macro defines a new minor mode whose name is *mode* (a symbol). It defines a command named *mode* to toggle the minor mode, with *doc* as its documentation string.

The toggle command takes one optional (prefix) argument. If called interactively with no argument it toggles the mode on or off. A positive prefix argument enables the mode, any other prefix argument disables it. From Lisp, an argument of `toggle` toggles the mode, whereas an omitted or `nil` argument enables the mode. This makes it easy to enable the minor mode in a major mode hook, for example. If *doc* is nil, the macro supplies a default documentation string explaining the above.

By default, it also defines a variable named *mode*, which is set to `t` or `nil` by enabling or disabling the mode. The variable is initialized to *init-value*. Except in unusual circumstances (see below), this value must be `nil`.

The string *lighter* says what to display in the mode line when the mode is enabled; if it is `nil`, the mode is not displayed in the mode line.

The optional argument *keymap* specifies the keymap for the minor mode. If non-`nil`, it should be a variable name (whose value is a keymap), a keymap, or an alist of the form

```
(key-sequence . definition)
```

where each *key-sequence* and *definition* are arguments suitable for passing to `define-key` (see ⟨undefined⟩ [Changing Key Bindings], page ⟨undefined⟩). If *keymap* is a keymap or an alist, this also defines the variable *mode*`-map`.

The above three arguments *init-value*, *lighter*, and *keymap* can be (partially) omitted when *keyword-args* are used. The *keyword-args* consist of keywords followed by corresponding values. A few keywords have special meanings:

`:group` *group*
: Custom group name to use in all generated `defcustom` forms. Defaults to *mode* without the possible trailing '`-mode`'. **Warning:** don't use this default group name unless you have written a `defgroup` to define that group properly. See ⟨undefined⟩ [Group Definitions], page ⟨undefined⟩.

`:global` *global*
: If non-`nil`, this specifies that the minor mode should be global rather than buffer-local. It defaults to `nil`.

  One of the effects of making a minor mode global is that the *mode* variable becomes a customization variable. Toggling it through the Customize interface turns the mode on and off, and its value can be saved for future Emacs sessions (see Section "Saving Customizations" in *The GNU Emacs Manual*. For the saved variable to work, you should ensure that the `define-minor-mode` form is evaluated each time Emacs starts; for packages that are not part of Emacs, the easiest way to do this is to specify a `:require` keyword.

`:init-value` *init-value*
: This is equivalent to specifying *init-value* positionally.

`:lighter` *lighter*
: This is equivalent to specifying *lighter* positionally.

`:keymap` *keymap*
: This is equivalent to specifying *keymap* positionally.

`:variable` *place*
: This replaces the default variable *mode*, used to store the state of the mode. If you specify this, the *mode* variable is not defined, and any *init-value* argument is unused. *place* can be a different named variable (which you must define yourself), or anything that can be used with the `setf` function (see ⟨undefined⟩ [Generalized Variables], page ⟨undefined⟩). *place* can also be a cons `(get . set)`, where *get* is an expression that returns the current state, and *set* is a function of one argument (a state) that sets it.

`:after-hook` *after-hook*
: This defines a single Lisp form which is evaluated after the mode hooks have run. It should not be quoted.

Any other keyword arguments are passed directly to the `defcustom` generated for the variable *mode*.

The command named *mode* first performs the standard actions such as setting the variable named *mode* and then executes the *body* forms, if any. It then runs the mode hook variable *mode*`-hook` and finishes by evaluating any form in `:after-hook`.

The initial value must be `nil` except in cases where (1) the mode is preloaded in Emacs, or (2) it is painless for loading to enable the mode even though the user did not request it. For instance, if the mode has no effect unless something else is enabled, and will always be loaded by that time, enabling it by default is harmless. But these are unusual circumstances. Normally, the initial value must be `nil`.

The name `easy-mmode-define-minor-mode` is an alias for this macro.

Here is an example of using `define-minor-mode`:

```
(define-minor-mode hungry-mode
  "Toggle Hungry mode.
Interactively with no argument, this command toggles the mode.
A positive prefix argument enables the mode, any other prefix
argument disables it.  From Lisp, argument omitted or nil enables
the mode, 'toggle' toggles the state.

When Hungry mode is enabled, the control delete key
gobbles all preceding whitespace except the last.
See the command \\[hungry-electric-delete]."
 ;; The initial value.
 nil
 ;; The indicator for the mode line.
 " Hungry"
 ;; The minor mode bindings.
 '(([C-backspace] . hungry-electric-delete))
 :group 'hunger)
```

This defines a minor mode named "Hungry mode", a command named `hungry-mode` to toggle it, a variable named `hungry-mode` which indicates whether the mode is enabled, and

a variable named `hungry-mode-map` which holds the keymap that is active when the mode is enabled. It initializes the keymap with a key binding for `C-DEL`. It puts the variable `hungry-mode` into custom group `hunger`. There are no *body* forms—many minor modes don't need any.

Here's an equivalent way to write it:

```
(define-minor-mode hungry-mode
  "Toggle Hungry mode.
...rest of documentation as before..."
 ;; The initial value.
 :init-value nil
 ;; The indicator for the mode line.
 :lighter " Hungry"
 ;; The minor mode bindings.
 :keymap
 '(([C-backspace] . hungry-electric-delete)
   ([C-M-backspace]
    . (lambda ()
        (interactive)
        (hungry-electric-delete t))))
 :group 'hunger)
```

`define-globalized-minor-mode` *global-mode mode turn-on keyword-args. . .* [Macro]

This defines a global toggle named *global-mode* whose meaning is to enable or disable the buffer-local minor mode *mode* in all buffers. To turn on the minor mode in a buffer, it uses the function *turn-on*; to turn off the minor mode, it calls `mode` with $-1$ as argument.

Globally enabling the mode also affects buffers subsequently created by visiting files, and buffers that use a major mode other than Fundamental mode; but it does not detect the creation of a new buffer in Fundamental mode.

This defines the customization option *global-mode* (see Chapter 33 [Customization], page 686), which can be toggled in the Customize interface to turn the minor mode on and off. As with `define-minor-mode`, you should ensure that the `define-globalized-minor-mode` form is evaluated each time Emacs starts, for example by providing a `:require` keyword.

Use `:group` *`group`* in *keyword-args* to specify the custom group for the mode variable of the global minor mode.

Generally speaking, when you define a globalized minor mode, you should also define a non-globalized version, so that people can use (or disable) it in individual buffers. This also allows them to disable a globally enabled minor mode in a specific major mode, by using that mode's hook.

## 20.4 Mode Line Format

Each Emacs window (aside from minibuffer windows) typically has a mode line at the bottom, which displays status information about the buffer displayed in the window. The mode line contains information about the buffer, such as its name, associated file, depth of recursive editing, and major and minor modes. A window can also have a *header line*, which is much like the mode line but appears at the top of the window.

This section describes how to control the contents of the mode line and header line. We include it in this chapter because much of the information displayed in the mode line relates to the enabled major and minor modes.

### 20.4.1 Mode Line Basics

The contents of each mode line are specified by the buffer-local variable `mode-line-format` (see Section 20.4.3 [Mode Line Top], page 421). This variable holds a *mode line construct*: a template that controls what is displayed on the buffer's mode line. The value of `header-line-format` specifies the buffer's header line in the same way. All windows for the same buffer use the same `mode-line-format` and `header-line-format`.

For efficiency, Emacs does not continuously recompute each window's mode line and header line. It does so when circumstances appear to call for it—for instance, if you change the window configuration, switch buffers, narrow or widen the buffer, scroll, or modify the buffer. If you alter any of the variables referenced by `mode-line-format` or `header-line-format` (see Section 20.4.4 [Mode Line Variables], page 422), or any other data structures that affect how text is displayed (see Chapter 11 [Display], page 111), you should use the function `force-mode-line-update` to update the display.

`force-mode-line-update` **&optional** *all* [Function]

This function forces Emacs to update the current buffer's mode line and header line, based on the latest values of all relevant variables, during its next redisplay cycle. If the optional argument *all* is non-`nil`, it forces an update for all mode lines and header lines.

This function also forces an update of the menu bar and frame title.

The selected window's mode line is usually displayed in a different color using the face `mode-line`. Other windows' mode lines appear in the face `mode-line-inactive` instead. See Section 11.12 [Faces], page 137.

### 20.4.2 The Data Structure of the Mode Line

The mode line contents are controlled by a data structure called a *mode line construct*, made up of lists, strings, symbols, and numbers kept in buffer-local variables. Each data type has a specific meaning for the mode line appearance, as described below. The same data structure is used for constructing frame titles (see Section 18.5 [Frame Titles], page 356) and header lines (see Section 20.4.7 [Header Lines], page 426).

A mode line construct may be as simple as a fixed string of text, but it usually specifies how to combine fixed strings with variables' values to construct the text. Many of these variables are themselves defined to have mode line constructs as their values.

Here are the meanings of various data types as mode line constructs:

*`string`* A string as a mode line construct appears verbatim except for `%`-*constructs* in it. These stand for substitution of other data; see Section 20.4.5 [%-Constructs], page 424.

If parts of the string have `face` properties, they control display of the text just as they would text in the buffer. Any characters which have no `face` properties are displayed, by default, in the face `mode-line` or `mode-line-inactive` (see Section "Standard Faces" in *The GNU Emacs Manual*). The `help-echo`

and `local-map` properties in *string* have special meanings. See Section 20.4.6 [Properties in Mode], page 425.

*symbol* A symbol as a mode line construct stands for its value. The value of *symbol* is used as a mode line construct, in place of *symbol*. However, the symbols `t` and `nil` are ignored, as is any symbol whose value is void.

There is one exception: if the value of *symbol* is a string, it is displayed verbatim: the `%`-constructs are not recognized.

Unless *symbol* is marked as "risky" (i.e., it has a non-`nil` `risky-local-variable` property), all text properties specified in *symbol*'s value are ignored. This includes the text properties of strings in *symbol*'s value, as well as all `:eval` and `:propertize` forms in it. (The reason for this is security: non-risky variables could be set automatically from file variables without prompting the user.)

`(`*string rest*`...)`
`(`*list rest*`...)`
A list whose first element is a string or list means to process all the elements recursively and concatenate the results. This is the most common form of mode line construct.

`(:eval` *form*`)`
A list whose first element is the symbol `:eval` says to evaluate *form*, and use the result as a string to display. Make sure this evaluation cannot load any files, as doing so could cause infinite recursion.

`(:propertize` *elt props*`...)`
A list whose first element is the symbol `:propertize` says to process the mode line construct *elt* recursively, then add the text properties specified by *props* to the result. The argument *props* should consist of zero or more pairs *text-property value*.

`(`*symbol then else*`)`
A list whose first element is a symbol that is not a keyword specifies a conditional. Its meaning depends on the value of *symbol*. If *symbol* has a non-`nil` value, the second element, *then*, is processed recursively as a mode line construct. Otherwise, the third element, *else*, is processed recursively. You may omit *else*; then the mode line construct displays nothing if the value of *symbol* is `nil` or void.

`(`*width rest*`...)`
A list whose first element is an integer specifies truncation or padding of the results of *rest*. The remaining elements *rest* are processed recursively as mode line constructs and concatenated together. When *width* is positive, the result is space filled on the right if its width is less than *width*. When *width* is negative, the result is truncated on the right to $-$*width* columns if its width exceeds $-$*width*.

For example, the usual way to show what percentage of a buffer is above the top of the window is to use a list like this: `(-3 "%p")`.

### 20.4.3 The Top Level of Mode Line Control

The variable in overall control of the mode line is `mode-line-format`.

`mode-line-format` [User Option]

The value of this variable is a mode line construct that controls the contents of the mode-line. It is always buffer-local in all buffers.

If you set this variable to `nil` in a buffer, that buffer does not have a mode line. (A window that is just one line tall also does not display a mode line.)

The default value of `mode-line-format` is designed to use the values of other variables such as `mode-line-position` and `mode-line-modes` (which in turn incorporates the values of the variables `mode-name` and `minor-mode-alist`). Very few modes need to alter `mode-line-format` itself. For most purposes, it is sufficient to alter some of the variables that `mode-line-format` either directly or indirectly refers to.

If you do alter `mode-line-format` itself, the new value should use the same variables that appear in the default value (see Section 20.4.4 [Mode Line Variables], page 422), rather than duplicating their contents or displaying the information in another fashion. This way, customizations made by the user or by Lisp programs (such as `display-time` and major modes) via changes to those variables remain effective.

Here is a hypothetical example of a `mode-line-format` that might be useful for Shell mode (in reality, Shell mode does not set `mode-line-format`):

```
(setq mode-line-format
  (list "-"
   'mode-line-mule-info
   'mode-line-modified
   'mode-line-frame-identification
   "%b--"
   ;; Note that this is evaluated while making the list.
   ;; It makes a mode line construct which is just a string.
   (getenv "HOST")
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
   '(:eval (mode-line-mode-name))
   'mode-line-process
   'minor-mode-alist
   "%n"
   ")%]--"
   '(which-func-mode ("" which-func-format "--"))
   '(line-number-mode "L%l--")
   '(column-number-mode "C%c--")
   '(-3 "%p")))
```

(The variables `line-number-mode`, `column-number-mode` and `which-func-mode` enable particular minor modes; as usual, these variable names are also the minor mode command names.)

### 20.4.4 Variables Used in the Mode Line

This section describes variables incorporated by the standard value of `mode-line-format` into the text of the mode line. There is nothing inherently special about these variables; any other variables could have the same effects on the mode line if the value of `mode-line-format` is changed to use them. However, various parts of Emacs set these variables on the understanding that they will control parts of the mode line; therefore, practically speaking, it is essential for the mode line to use them.

`mode-line-mule-info` [Variable]
: This variable holds the value of the mode line construct that displays information about the language environment, buffer coding system, and current input method. See ⟨undefined⟩ [Non-ASCII Characters], page ⟨undefined⟩.

`mode-line-modified` [Variable]
: This variable holds the value of the mode line construct that displays whether the current buffer is modified. Its default value displays '`**`' if the buffer is modified, '`--`' if the buffer is not modified, '`%%`' if the buffer is read only, and '`%*`' if the buffer is read only and modified.

  Changing this variable does not force an update of the mode line.

`mode-line-frame-identification` [Variable]
: This variable identifies the current frame. Its default value displays `" "` if you are using a window system which can show multiple frames, or `"-%F "` on an ordinary terminal which shows only one frame at a time.

`mode-line-buffer-identification` [Variable]
: This variable identifies the buffer being displayed in the window. Its default value displays the buffer name, padded with spaces to at least 12 columns.

`mode-line-position` [User Option]
: This variable indicates the position in the buffer. Its default value displays the buffer percentage and, optionally, the buffer size, the line number and the column number.

`vc-mode` [Variable]
: The variable `vc-mode`, buffer-local in each buffer, records whether the buffer's visited file is maintained with version control, and, if so, which kind. Its value is a string that appears in the mode line, or `nil` for no version control.

`mode-line-modes` [User Option]
: This variable displays the buffer's major and minor modes. Its default value also displays the recursive editing level, information on the process status, and whether narrowing is in effect.

`mode-line-remote` [Variable]
: This variable is used to show whether `default-directory` for the current buffer is remote.

`mode-line-client` [Variable]
: This variable is used to identify `emacsclient` frames.

The following three variables are used in `mode-line-modes`:

`mode-name` [Variable]
: This buffer-local variable holds the "pretty" name of the current buffer's major mode. Each major mode should set this variable so that the mode name will appear in the mode line. The value does not have to be a string, but can use any of the data types valid in a mode-line construct (see Section 20.4.2 [Mode Line Data], page 419). To compute the string that will identify the mode name in the mode line, use `format-mode-line` (see Section 20.4.8 [Emulating Mode Line], page 426).

`mode-line-process` [Variable]
: This buffer-local variable contains the mode line information on process status in modes used for communicating with subprocesses. It is displayed immediately following the major mode name, with no intervening space. For example, its value in the '`*shell*`' buffer is `(":%s")`, which allows the shell to display its status along with the major mode as: '`(Shell:run)`'. Normally this variable is `nil`.

`minor-mode-alist` [Variable]
: This variable holds an association list whose elements specify how the mode line should indicate that a minor mode is active. Each element of the `minor-mode-alist` should be a two-element list:

  ```
  (minor-mode-variable mode-line-string)
  ```

  More generally, *mode-line-string* can be any mode line construct. It appears in the mode line when the value of *minor-mode-variable* is non-`nil`, and not otherwise. These strings should begin with spaces so that they don't run together. Conventionally, the *minor-mode-variable* for a specific mode is set to a non-`nil` value when that minor mode is activated.

  `minor-mode-alist` itself is not buffer-local. Each variable mentioned in the alist should be buffer-local if its minor mode can be enabled separately in each buffer.

`global-mode-string` [Variable]
: This variable holds a mode line construct that, by default, appears in the mode line just after the `which-func-mode` minor mode if set, else after `mode-line-modes`. The command `display-time` sets `global-mode-string` to refer to the variable `display-time-string`, which holds a string containing the time and load information.

  The '`%M`' construct substitutes the value of `global-mode-string`, but that is obsolete, since the variable is included in the mode line from `mode-line-format`.

Here is a simplified version of the default value of `mode-line-format`. The real default value also specifies addition of text properties.

```
("-"
 mode-line-mule-info
 mode-line-modified
 mode-line-frame-identification
 mode-line-buffer-identification
```

```
"   "
mode-line-position
(vc-mode vc-mode)
"   "
mode-line-modes
(which-func-mode ("" which-func-format "--"))
(global-mode-string ("--" global-mode-string))
"-%-")
```

### 20.4.5 %-Constructs in the Mode Line

Strings used as mode line constructs can use certain %-constructs to substitute various kinds of data. The following is a list of the defined %-constructs, and what they mean.

In any construct except '%%', you can add a decimal integer after the '%' to specify a minimum field width. If the width is less, the field is padded to that width. Purely numeric constructs ('c', 'i', 'I', and 'l') are padded by inserting spaces to the left, and others are padded by inserting spaces to the right.

`%b` The current buffer name, obtained with the `buffer-name` function. See Section 16.3 [Buffer Names], page 275.

`%c` The current column number of point.

`%e` When Emacs is nearly out of memory for Lisp objects, a brief message saying so. Otherwise, this is empty.

`%f` The visited file name, obtained with the `buffer-file-name` function. See Section 16.4 [Buffer File Name], page 276.

`%F` The title (only on a window system) or the name of the selected frame. See Section 18.3.3.1 [Basic Parameters], page 346.

`%i` The size of the accessible part of the current buffer; basically `(- (point-max) (point-min))`.

`%I` Like '`%i`', but the size is printed in a more readable way by using '`k`' for 10^3, '`M`' for 10^6, '`G`' for 10^9, etc., to abbreviate.

`%l` The current line number of point, counting within the accessible portion of the buffer.

`%n` '`Narrow`' when narrowing is in effect; nothing otherwise (see `narrow-to-region` in ⟨undefined⟩ [Narrowing], page ⟨undefined⟩).

`%p` The percentage of the buffer text above the **top** of window, or '`Top`', '`Bottom`' or '`All`'. Note that the default mode line construct truncates this to three characters.

`%P` The percentage of the buffer text that is above the **bottom** of the window (which includes the text visible in the window, as well as the text above the top), plus '`Top`' if the top of the buffer is visible on screen; or '`Bottom`' or '`All`'.

`%s` The status of the subprocess belonging to the current buffer, obtained with `process-status`. See ⟨undefined⟩ [Process Information], page ⟨undefined⟩.

`%t` Whether the visited file is a text file or a binary file. This is a meaningful distinction only on certain operating systems (see ⟨undefined⟩ [MS-DOS File Types], page ⟨undefined⟩).

`%z` The mnemonics of keyboard, terminal, and buffer coding systems.

`%Z` Like '`%z`', but including the end-of-line format.

`%*` '`%`' if the buffer is read only (see `buffer-read-only`);
'`*`' if the buffer is modified (see `buffer-modified-p`);
'`-`' otherwise. See Section 16.5 [Buffer Modification], page 278.

`%+` '`*`' if the buffer is modified (see `buffer-modified-p`);
'`%`' if the buffer is read only (see `buffer-read-only`);
'`-`' otherwise. This differs from '`%*`' only for a modified read-only buffer. See Section 16.5 [Buffer Modification], page 278.

`%&` '`*`' if the buffer is modified, and '`-`' otherwise.

`%[` An indication of the depth of recursive editing levels (not counting minibuffer levels): one '`[`' for each editing level. See Section 2.13 [Recursive Editing], page 51.

`%]` One '`]`' for each recursive editing level (not counting minibuffer levels).

`%-` Dashes sufficient to fill the remainder of the mode line.

`%%` The character '`%`'—this is how to include a literal '`%`' in a string in which `%`-constructs are allowed.

The following two `%`-constructs are still supported, but they are obsolete, since you can get the same results with the variables `mode-name` and `global-mode-string`.

`%m` The value of `mode-name`.

`%M` The value of `global-mode-string`.

### 20.4.6 Properties in the Mode Line

Certain text properties are meaningful in the mode line. The `face` property affects the appearance of text; the `help-echo` property associates help strings with the text, and `local-map` can make the text mouse-sensitive.

There are four ways to specify text properties for text in the mode line:

1. Put a string with a text property directly into the mode line data structure.
2. Put a text property on a mode line %-construct such as '`%12b`'; then the expansion of the %-construct will have that same text property.
3. Use a `(:propertize` *`elt props`*`...)` construct to give *elt* a text property specified by *props*.
4. Use a list containing `:eval` *`form`* in the mode line data structure, and make *form* evaluate to a string that has a text property.

You can use the `local-map` property to specify a keymap. This keymap only takes real effect for mouse clicks; binding character keys and function keys to it has no effect, since it is impossible to move point into the mode line.

When the mode line refers to a variable which does not have a non-`nil` `risky-local-variable` property, any text properties given or specified within that variable's values are ignored. This is because such properties could otherwise specify functions to be called, and those functions could come from file local variables.

### 20.4.7 Window Header Lines

A window can have a *header line* at the top, just as it can have a mode line at the bottom. The header line feature works just like the mode line feature, except that it's controlled by `header-line-format`:

`header-line-format` [Variable]

This variable, local in every buffer, specifies how to display the header line, for windows displaying the buffer. The format of the value is the same as for `mode-line-format` (see Section 20.4.2 [Mode Line Data], page 419). It is normally `nil`, so that ordinary buffers have no header line.

A window that is just one line tall never displays a header line. A window that is two lines tall cannot display both a mode line and a header line at once; if it has a mode line, then it does not display a header line.

### 20.4.8 Emulating Mode Line Formatting

You can use the function `format-mode-line` to compute the text that would appear in a mode line or header line based on a certain mode line construct.

`format-mode-line` *format* **&optional** *face window buffer* [Function]

This function formats a line of text according to *format* as if it were generating the mode line for *window*, but it also returns the text as a string. The argument *window* defaults to the selected window. If *buffer* is non-`nil`, all the information used is taken from *buffer*; by default, it comes from *window*'s buffer.

The value string normally has text properties that correspond to the faces, keymaps, etc., that the mode line would have. Any character for which no `face` property is specified by *format* gets a default value determined by *face*. If *face* is `t`, that stands for either `mode-line` if *window* is selected, otherwise `mode-line-inactive`. If *face* is `nil` or omitted, that stands for the default face. If *face* is an integer, the value returned by this function will have no text properties.

You can also specify other valid faces as the value of *face*. If specified, that face provides the `face` property for characters whose face is not specified by *format*.

Note that using `mode-line`, `mode-line-inactive`, or `header-line` as *face* will actually redisplay the mode line or the header line, respectively, using the current definitions of the corresponding face, in addition to returning the formatted string. (Other faces do not cause redisplay.)

For example, `(format-mode-line header-line-format)` returns the text that would appear in the selected window's header line (`""` if it has no header line). `(format-mode-line header-line-format 'header-line)` returns the same text, with each character carrying the face that it will have in the header line itself, and also redraws the header line.

## 20.5 Imenu

*Imenu* is a feature that lets users select a definition or section in the buffer, from a menu which lists all of them, to go directly to that location in the buffer. Imenu works by constructing a buffer index which lists the names and buffer positions of the definitions, or other named portions of the buffer; then the user can choose one of them and move point to it. Major modes can add a menu bar item to use Imenu using `imenu-add-to-menubar`.

`imenu-add-to-menubar` *name* [Command]
: This function defines a local menu bar item named *name* to run Imenu.

The user-level commands for using Imenu are described in the Emacs Manual (see Section "Imenu" in *the Emacs Manual*). This section explains how to customize Imenu's method of finding definitions or buffer portions for a particular major mode.

The usual and simplest way is to set the variable `imenu-generic-expression`:

`imenu-generic-expression` [Variable]
: This variable, if non-`nil`, is a list that specifies regular expressions for finding definitions for Imenu. Simple elements of `imenu-generic-expression` look like this:

  ```
  (menu-title regexp index)
  ```

  Here, if *menu-title* is non-`nil`, it says that the matches for this element should go in a submenu of the buffer index; *menu-title* itself specifies the name for the submenu. If *menu-title* is `nil`, the matches for this element go directly in the top level of the buffer index.

  The second item in the list, *regexp*, is a regular expression (see ⟨undefined⟩ [Regular Expressions], page ⟨undefined⟩); anything in the buffer that it matches is considered a definition, something to mention in the buffer index. The third item, *index*, is a non-negative integer that indicates which subexpression in *regexp* matches the definition's name.

  An element can also look like this:

  ```
  (menu-title regexp index function arguments...)
  ```

  Each match for this element creates an index item, and when the index item is selected by the user, it calls *function* with arguments consisting of the item name, the buffer position, and *arguments*.

  For Emacs Lisp mode, `imenu-generic-expression` could look like this:

  ```
  ((nil "^\\s-*(def\\(un\\|subst\\|macro\\|advice\\)\
  \\s-+\\([-A-Za-z0-9+]+\\)" 2)
   ("*Vars*" "^\\s-*(def\\(var\\|const\\)\
  \\s-+\\([-A-Za-z0-9+]+\\)" 2)
   ("*Types*"
    "^\\s-*\
  (def\\(type\\|struct\\|class\\|ine-condition\\)\
  \\s-+\\([-A-Za-z0-9+]+\\)" 2))
  ```

  Setting this variable makes it buffer-local in the current buffer.

`imenu-case-fold-search` [Variable]

This variable controls whether matching against the regular expressions in the value of `imenu-generic-expression` is case-sensitive: `t`, the default, means matching should ignore case.

Setting this variable makes it buffer-local in the current buffer.

`imenu-syntax-alist` [Variable]

This variable is an alist of syntax table modifiers to use while processing `imenu-generic-expression`, to override the syntax table of the current buffer. Each element should have this form:

```
(characters . syntax-description)
```

The CAR, *characters*, can be either a character or a string. The element says to give that character or characters the syntax specified by *syntax-description*, which is passed to `modify-syntax-entry` (see ⟨undefined⟩ [Syntax Table Functions], page ⟨undefined⟩).

This feature is typically used to give word syntax to characters which normally have symbol syntax, and thus to simplify `imenu-generic-expression` and speed up matching. For example, Fortran mode uses it this way:

```
(setq imenu-syntax-alist '(("_$" . "w")))
```

The `imenu-generic-expression` regular expressions can then use '`\\sw+`' instead of '`\\(\\sw\\|\\s_\\)+`'. Note that this technique may be inconvenient when the mode needs to limit the initial character of a name to a smaller set of characters than are allowed in the rest of a name.

Setting this variable makes it buffer-local in the current buffer.

Another way to customize Imenu for a major mode is to set the variables `imenu-prev-index-position-function` and `imenu-extract-index-name-function`:

`imenu-prev-index-position-function` [Variable]

If this variable is non-`nil`, its value should be a function that finds the next "definition" to put in the buffer index, scanning backward in the buffer from point. It should return `nil` if it doesn't find another "definition" before point. Otherwise it should leave point at the place it finds a "definition" and return any non-`nil` value.

Setting this variable makes it buffer-local in the current buffer.

`imenu-extract-index-name-function` [Variable]

If this variable is non-`nil`, its value should be a function to return the name for a definition, assuming point is in that definition as the `imenu-prev-index-position-function` function would leave it.

Setting this variable makes it buffer-local in the current buffer.

The last way to customize Imenu for a major mode is to set the variable `imenu-create-index-function`:

`imenu-create-index-function` [Variable]

This variable specifies the function to use for creating a buffer index. The function should take no arguments, and return an index alist for the current buffer. It is called within `save-excursion`, so where it leaves point makes no difference.

The index alist can have three types of elements. Simple elements look like this:

```
(index-name . index-position)
```

Selecting a simple element has the effect of moving to position *index-position* in the buffer. Special elements look like this:

```
(index-name index-position function arguments...)
```

Selecting a special element performs:

```
(funcall function
         index-name index-position arguments...)
```

A nested sub-alist element looks like this:

```
(menu-title sub-alist)
```

It creates the submenu *menu-title* specified by *sub-alist*.

The default value of `imenu-create-index-function` is `imenu-default-create-index-function`. This function calls the value of `imenu-prev-index-position-function` and the value of `imenu-extract-index-name-function` to produce the index alist. However, if either of these two variables is `nil`, the default function uses `imenu-generic-expression` instead.

Setting this variable makes it buffer-local in the current buffer.

## 20.6 Font Lock Mode

*Font Lock mode* is a buffer-local minor mode that automatically attaches `face` properties to certain parts of the buffer based on their syntactic role. How it parses the buffer depends on the major mode; most major modes define syntactic criteria for which faces to use in which contexts. This section explains how to customize Font Lock for a particular major mode.

Font Lock mode finds text to highlight in two ways: through syntactic parsing based on the syntax table, and through searching (usually for regular expressions). Syntactic fontification happens first; it finds comments and string constants and highlights them. Search-based fontification happens second.

### 20.6.1 Font Lock Basics

There are several variables that control how Font Lock mode highlights text. But major modes should not set any of these variables directly. Instead, they should set `font-lock-defaults` as a buffer-local variable. The value assigned to this variable is used, if and when Font Lock mode is enabled, to set all the other variables.

`font-lock-defaults` [Variable]

This variable is set by major modes to specify how to fontify text in that mode. It automatically becomes buffer-local when set. If its value is `nil`, Font Lock mode does no highlighting, and you can use the '`Faces`' menu (under '`Edit`' and then '`Text Properties`' in the menu bar) to assign faces explicitly to text in the buffer.

If non-`nil`, the value should look like this:

```
(keywords [keywords-only [case-fold
 [syntax-alist [syntax-begin other-vars...]]]])
```

The first element, *keywords*, indirectly specifies the value of `font-lock-keywords` which directs search-based fontification. It can be a symbol, a variable or a function whose value is the list to use for `font-lock-keywords`. It can also be a list of several such symbols, one for each possible level of fontification. The first symbol specifies the '`mode default`' level of fontification, the next symbol level 1 fontification, the next level 2, and so on. The '`mode default`' level is normally the same as level 1. It is used when `font-lock-maximum-decoration` has a `nil` value. See Section 20.6.5 [Levels of Font Lock], page 436.

The second element, *keywords-only*, specifies the value of the variable `font-lock-keywords-only`. If this is omitted or `nil`, syntactic fontification (of strings and comments) is also performed. If this is non-`nil`, syntactic fontification is not performed. See Section 20.6.8 [Syntactic Font Lock], page 437.

The third element, *case-fold*, specifies the value of `font-lock-keywords-case-fold-search`. If it is non-`nil`, Font Lock mode ignores case during search-based fontification.

If the fourth element, *syntax-alist*, is non-`nil`, it should be a list of cons cells of the form `(`*`char-or-string . string`*`)`. These are used to set up a syntax table for syntactic fontification; the resulting syntax table is stored in `font-lock-syntax-table`. If *syntax-alist* is omitted or `nil`, syntactic fontification uses the syntax table returned by the `syntax-table` function. See ⟨undefined⟩ [Syntax Table Functions], page ⟨undefined⟩.

The fifth element, *syntax-begin*, specifies the value of `font-lock-beginning-of-syntax-function`. We recommend setting this variable to `nil` and using `syntax-begin-function` instead.

All the remaining elements (if any) are collectively called *other-vars*. Each of these elements should have the form `(`*`variable . value`*`)`—which means, make *variable* buffer-local and then set it to *value*. You can use these *other-vars* to set other variables that affect fontification, aside from those you can control with the first five elements. See Section 20.6.4 [Other Font Lock Variables], page 435.

If your mode fontifies text explicitly by adding `font-lock-face` properties, it can specify `(nil t)` for `font-lock-defaults` to turn off all automatic fontification. However, this is not required; it is possible to fontify some things using `font-lock-face` properties and set up automatic fontification for other parts of the text.

### 20.6.2 Search-based Fontification

The variable which directly controls search-based fontification is `font-lock-keywords`, which is typically specified via the *keywords* element in `font-lock-defaults`.

`font-lock-keywords` [Variable]

The value of this variable is a list of the keywords to highlight. Lisp programs should not set this variable directly. Normally, the value is automatically set by Font Lock mode, using the *keywords* element in `font-lock-defaults`. The value can also be altered using the functions `font-lock-add-keywords` and `font-lock-remove-keywords` (see Section 20.6.3 [Customizing Keywords], page 434).

Each element of `font-lock-keywords` specifies how to find certain cases of text, and how to highlight those cases. Font Lock mode processes the elements of `font-lock-keywords` one by one, and for each element, it finds and handles all matches. Ordinarily, once part of the text has been fontified already, this cannot be overridden by a subsequent match in the same text; but you can specify different behavior using the *override* element of a *subexp-highlighter*.

Each element of `font-lock-keywords` should have one of these forms:

*regexp* Highlight all matches for *regexp* using `font-lock-keyword-face`. For example,

```
;; Highlight occurrences of the word 'foo'
;; using font-lock-keyword-face.
"\\<foo\\>"
```

Be careful when composing these regular expressions; a poorly written pattern can dramatically slow things down! The function `regexp-opt` (see ⟨undefined⟩ [Regexp Functions], page ⟨undefined⟩) is useful for calculating optimal regular expressions to match several keywords.

*function* Find text by calling *function*, and highlight the matches it finds using `font-lock-keyword-face`.

When *function* is called, it receives one argument, the limit of the search; it should begin searching at point, and not search beyond the limit. It should return non-`nil` if it succeeds, and set the match data to describe the match that was found. Returning `nil` indicates failure of the search.

Fontification will call *function* repeatedly with the same limit, and with point where the previous invocation left it, until *function* fails. On failure, *function* need not reset point in any particular way.

`(`*matcher* `.` *subexp*`)`
In this kind of element, *matcher* is either a regular expression or a function, as described above. The CDR, *subexp*, specifies which subexpression of *matcher* should be highlighted (instead of the entire text that *matcher* matched).

```
;; Highlight the 'bar' in each occurrence of 'fubar',
;; using font-lock-keyword-face.
("fu\\(bar\\)" . 1)
```

If you use `regexp-opt` to produce the regular expression *matcher*, you can use `regexp-opt-depth` (see ⟨undefined⟩ [Regexp Functions], page ⟨undefined⟩) to calculate the value for *subexp*.

`(`*matcher* `.` *facespec*`)`
In this kind of element, *facespec* is an expression whose value specifies the face to use for highlighting. In the simplest case, *facespec* is a Lisp variable (a symbol) whose value is a face name.

```
;; Highlight occurrences of 'fubar',
;; using the face which is the value of fubar-face.
("fubar" . fubar-face)
```

However, *facespec* can also evaluate to a list of this form:

```
(face face prop1 val1 prop2 val2...)
```

to specify the face *face* and various additional text properties to put on the text that matches. If you do this, be sure to add the other text property names that you set in this way to the value of `font-lock-extra-managed-props` so that the properties will also be cleared out when they are no longer appropriate. Alternatively, you can set the variable `font-lock-unfontify-region-function` to a function that clears these properties. See Section 20.6.4 [Other Font Lock Variables], page 435.

`(`*matcher* `.` *subexp-highlighter*`)`

In this kind of element, *subexp-highlighter* is a list which specifies how to highlight matches found by *matcher*. It has the form:

```
(subexp facespec [override [laxmatch]])
```

The CAR, *subexp*, is an integer specifying which subexpression of the match to fontify (0 means the entire matching text). The second subelement, *facespec*, is an expression whose value specifies the face, as described above.

The last two values in *subexp-highlighter*, *override* and *laxmatch*, are optional flags. If *override* is `t`, this element can override existing fontification made by previous elements of `font-lock-keywords`. If it is `keep`, then each character is fontified if it has not been fontified already by some other element. If it is `prepend`, the face specified by *facespec* is added to the beginning of the `font-lock-face` property. If it is `append`, the face is added to the end of the `font-lock-face` property.

If *laxmatch* is non-`nil`, it means there should be no error if there is no subexpression numbered *subexp* in *matcher*. Obviously, fontification of the subexpression numbered *subexp* will not occur. However, fontification of other subexpressions (and other regexps) will continue. If *laxmatch* is `nil`, and the specified subexpression is missing, then an error is signaled which terminates search-based fontification.

Here are some examples of elements of this kind, and what they do:

```
;; Highlight occurrences of either 'foo' or 'bar', using
;; foo-bar-face, even if they have already been highlighted.
;; foo-bar-face should be a variable whose value is a face.
("foo\\|bar" 0 foo-bar-face t)

;; Highlight the first subexpression within each occurrence
;; that the function fubar-match finds,
;; using the face which is the value of fubar-face.
(fubar-match 1 fubar-face)
```

`(`*matcher* `.` *anchored-highlighter*`)`

In this kind of element, *anchored-highlighter* specifies how to highlight text that follows a match found by *matcher*. So a match found by *matcher* acts as the anchor for further searches specified by *anchored-highlighter*. *anchored-highlighter* is a list of the following form:

```
(anchored-matcher pre-form post-form
                  subexp-highlighters...)
```

Here, *anchored-matcher*, like *matcher*, is either a regular expression or a function. After a match of *matcher* is found, point is at the end of the match. Now, Font Lock evaluates the form *pre-form*. Then it searches for matches of *anchored-matcher* and uses *subexp-highlighters* to highlight these. A *subexp-highlighter* is as described above. Finally, Font Lock evaluates *post-form*.

The forms *pre-form* and *post-form* can be used to initialize before, and cleanup after, *anchored-matcher* is used. Typically, *pre-form* is used to move point to some position relative to the match of *matcher*, before starting with *anchored-matcher*. *post-form* might be used to move back, before resuming with *matcher*.

After Font Lock evaluates *pre-form*, it does not search for *anchored-matcher* beyond the end of the line. However, if *pre-form* returns a buffer position that is greater than the position of point after *pre-form* is evaluated, then the position returned by *pre-form* is used as the limit of the search instead. It is generally a bad idea to return a position greater than the end of the line; in other words, the *anchored-matcher* search should not span lines.

For example,

```
;; Highlight occurrences of the word 'item' following
;; an occurrence of the word 'anchor' (on the same line)
;; in the value of item-face.
("\\<anchor\\>" "\\<item\\>" nil nil (0 item-face))
```

Here, *pre-form* and *post-form* are `nil`. Therefore searching for '`item`' starts at the end of the match of '`anchor`', and searching for subsequent instances of '`anchor`' resumes from where searching for '`item`' concluded.

`(`*matcher highlighters*`...)`

This sort of element specifies several *highlighter* lists for a single *matcher*. A *highlighter* list can be of the type *subexp-highlighter* or *anchored-highlighter* as described above.

For example,

```
;; Highlight occurrences of the word 'anchor' in the value
;; of anchor-face, and subsequent occurrences of the word
;; 'item' (on the same line) in the value of item-face.
("\\<anchor\\>" (0 anchor-face)
                ("\\<item\\>" nil nil (0 item-face)))
```

`(eval .` *form*`)`

Here *form* is an expression to be evaluated the first time this value of `font-lock-keywords` is used in a buffer. Its value should have one of the forms described in this table.

**Warning:** Do not design an element of `font-lock-keywords` to match text which spans lines; this does not work reliably. For details, see See Section 20.6.9 [Multiline Font Lock], page 438.

You can use *case-fold* in `font-lock-defaults` to specify the value of `font-lock-keywords-case-fold-search` which says whether search-based fontification should be case-insensitive.

`font-lock-keywords-case-fold-search` [Variable]

Non-`nil` means that regular expression matching for the sake of `font-lock-keywords` should be case-insensitive.

### 20.6.3 Customizing Search-Based Fontification

You can use `font-lock-add-keywords` to add additional search-based fontification rules to a major mode, and `font-lock-remove-keywords` to remove rules.

`font-lock-add-keywords` *mode keywords* **&optional** *how* [Function]

This function adds highlighting *keywords*, for the current buffer or for major mode *mode*. The argument *keywords* should be a list with the same format as the variable `font-lock-keywords`.

If *mode* is a symbol which is a major mode command name, such as `c-mode`, the effect is that enabling Font Lock mode in *mode* will add *keywords* to `font-lock-keywords`. Calling with a non-`nil` value of *mode* is correct only in your '`~/.emacs`' file.

If *mode* is `nil`, this function adds *keywords* to `font-lock-keywords` in the current buffer. This way of calling `font-lock-add-keywords` is usually used in mode hook functions.

By default, *keywords* are added at the beginning of `font-lock-keywords`. If the optional argument *how* is `set`, they are used to replace the value of `font-lock-keywords`. If *how* is any other non-`nil` value, they are added at the end of `font-lock-keywords`.

Some modes provide specialized support you can use in additional highlighting patterns. See the variables `c-font-lock-extra-types`, `c++-font-lock-extra-types`, and `java-font-lock-extra-types`, for example.

**Warning:** Major mode commands must not call `font-lock-add-keywords` under any circumstances, either directly or indirectly, except through their mode hooks. (Doing so would lead to incorrect behavior for some minor modes.) They should set up their rules for search-based fontification by setting `font-lock-keywords`.

`font-lock-remove-keywords` *mode keywords* [Function]

This function removes *keywords* from `font-lock-keywords` for the current buffer or for major mode *mode*. As in `font-lock-add-keywords`, *mode* should be a major mode command name or `nil`. All the caveats and requirements for `font-lock-add-keywords` apply here too.

For example, the following code adds two fontification patterns for C mode: one to fontify the word '`FIXME`', even in comments, and another to fontify the words '`and`', '`or`' and '`not`' as keywords.

```
(font-lock-add-keywords 'c-mode
 '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
   ("\\<\\(and\\|or\\|not\\)\\>" . font-lock-keyword-face)))
```

This example affects only C mode proper. To add the same patterns to C mode *and* all modes derived from it, do this instead:

```
(add-hook 'c-mode-hook
 (lambda ()
  (font-lock-add-keywords nil
   '(("\\<\\(FIXME\\):" 1 font-lock-warning-face prepend)
     ("\\<\\(and\\|or\\|not\\)\\>" .
      font-lock-keyword-face)))))
```

### 20.6.4 Other Font Lock Variables

This section describes additional variables that a major mode can set by means of *other-vars* in `font-lock-defaults` (see Section 20.6.1 [Font Lock Basics], page 429).

`font-lock-mark-block-function` [Variable]

If this variable is non-`nil`, it should be a function that is called with no arguments, to choose an enclosing range of text for refontification for the command `M-o M-o` (`font-lock-fontify-block`).

The function should report its choice by placing the region around it. A good choice is a range of text large enough to give proper results, but not too large so that refontification becomes slow. Typical values are `mark-defun` for programming modes or `mark-paragraph` for textual modes.

`font-lock-extra-managed-props` [Variable]

This variable specifies additional properties (other than `font-lock-face`) that are being managed by Font Lock mode. It is used by `font-lock-default-unfontify-region`, which normally only manages the `font-lock-face` property. If you want Font Lock to manage other properties as well, you must specify them in a *facespec* in `font-lock-keywords` as well as add them to this list. See Section 20.6.2 [Search-based Fontification], page 430.

`font-lock-fontify-buffer-function` [Variable]

Function to use for fontifying the buffer. The default value is `font-lock-default-fontify-buffer`.

`font-lock-unfontify-buffer-function` [Variable]

Function to use for unfontifying the buffer. This is used when turning off Font Lock mode. The default value is `font-lock-default-unfontify-buffer`.

`font-lock-fontify-region-function` [Variable]

Function to use for fontifying a region. It should take two arguments, the beginning and end of the region, and an optional third argument *verbose*. If *verbose* is non-`nil`, the function should print status messages. The default value is `font-lock-default-fontify-region`.

`font-lock-unfontify-region-function` [Variable]

Function to use for unfontifying a region. It should take two arguments, the beginning and end of the region. The default value is `font-lock-default-unfontify-region`.

`jit-lock-register` *function* **&optional** *contextual* [Function]

This function tells Font Lock mode to run the Lisp function *function* any time it has to fontify or refontify part of the current buffer. It calls *function* before calling the default fontification functions, and gives it two arguments, *start* and *end*, which specify the region to be fontified or refontified.

The optional argument *contextual*, if non-`nil`, forces Font Lock mode to always refontify a syntactically relevant part of the buffer, and not just the modified lines. This argument can usually be omitted.

`jit-lock-unregister` *function* [Function]
: If *function* was previously registered as a fontification function using `jit-lock-register`, this function unregisters it.

### 20.6.5 Levels of Font Lock

Some major modes offer three different levels of fontification. You can define multiple levels by using a list of symbols for *keywords* in `font-lock-defaults`. Each symbol specifies one level of fontification; it is up to the user to choose one of these levels, normally by setting `font-lock-maximum-decoration` (see Section "Font Lock" in *the GNU Emacs Manual*). The chosen level's symbol value is used to initialize `font-lock-keywords`.

Here are the conventions for how to define the levels of fontification:

- Level 1: highlight function declarations, file directives (such as include or import directives), strings and comments. The idea is speed, so only the most important and top-level components are fontified.
- Level 2: in addition to level 1, highlight all language keywords, including type names that act like keywords, as well as named constant values. The idea is that all keywords (either syntactic or semantic) should be fontified appropriately.
- Level 3: in addition to level 2, highlight the symbols being defined in function and variable declarations, and all builtin function names, wherever they appear.

### 20.6.6 Precalculated Fontification

Some major modes such as `list-buffers` and `occur` construct the buffer text programmatically. The easiest way for them to support Font Lock mode is to specify the faces of text when they insert the text in the buffer.

The way to do this is to specify the faces in the text with the special text property `font-lock-face` (see Section 22.19.4 [Special Properties], page 494). When Font Lock mode is enabled, this property controls the display, just like the `face` property. When Font Lock mode is disabled, `font-lock-face` has no effect on the display.

It is ok for a mode to use `font-lock-face` for some text and also use the normal Font Lock machinery. But if the mode does not use the normal Font Lock machinery, it should not set the variable `font-lock-defaults`.

### 20.6.7 Faces for Font Lock

Font Lock mode can highlight using any face, but Emacs defines several faces specifically for Font Lock to use to highlight text. These *Font Lock faces* are listed below. They can also be used by major modes for syntactic highlighting outside of Font Lock mode (see Section 20.2.1 [Major Mode Conventions], page 399).

Each of these symbols is both a face name, and a variable whose default value is the symbol itself. Thus, the default value of `font-lock-comment-face` is `font-lock-comment-face`.

The faces are listed with descriptions of their typical usage, and in order of greater to lesser "prominence". If a mode's syntactic categories do not fit well with the usage descriptions, the faces can be assigned using the ordering as a guide.

`font-lock-warning-face`
: for a construct that is peculiar, or that greatly changes the meaning of other text, like '`;;;###autoload`' in Emacs Lisp and '`#error`' in C.

`font-lock-function-name-face`
: for the name of a function being defined or declared.

`font-lock-variable-name-face`
: for the name of a variable being defined or declared.

`font-lock-keyword-face`
: for a keyword with special syntactic significance, like '`for`' and '`if`' in C.

`font-lock-comment-face`
: for comments.

`font-lock-comment-delimiter-face`
: for comments delimiters, like '`/*`' and '`*/`' in C. On most terminals, this inherits from `font-lock-comment-face`.

`font-lock-type-face`
: for the names of user-defined data types.

`font-lock-constant-face`
: for the names of constants, like '`NULL`' in C.

`font-lock-builtin-face`
: for the names of built-in functions.

`font-lock-preprocessor-face`
: for preprocessor commands. This inherits, by default, from `font-lock-builtin-face`.

`font-lock-string-face`
: for string constants.

`font-lock-doc-face`
: for documentation strings in the code. This inherits, by default, from `font-lock-string-face`.

`font-lock-negation-char-face`
: for easily-overlooked negation characters.

### 20.6.8 Syntactic Font Lock

Syntactic fontification uses a syntax table (see ⟨undefined⟩ [Syntax Tables], page ⟨undefined⟩) to find and highlight syntactically relevant text. If enabled, it runs prior to search-based fontification. The variable `font-lock-syntactic-face-function`, documented below, determines which syntactic constructs to highlight. There are several variables that affect syntactic fontification; you should set them by means of `font-lock-defaults` (see Section 20.6.1 [Font Lock Basics], page 429).

Whenever Font Lock mode performs syntactic fontification on a stretch of text, it first calls the function specified by `syntax-propertize-function`. Major modes can use this to apply `syntax-table` text properties to override the buffer's syntax table in special cases. See ⟨undefined⟩ [Syntax Properties], page ⟨undefined⟩.

`font-lock-keywords-only` [Variable]
: If the value of this variable is non-`nil`, Font Lock does not do syntactic fontification, only search-based fontification based on `font-lock-keywords`. It is normally set by Font Lock mode based on the *keywords-only* element in `font-lock-defaults`.

`font-lock-syntax-table` [Variable]
: This variable holds the syntax table to use for fontification of comments and strings. It is normally set by Font Lock mode based on the *syntax-alist* element in `font-lock-defaults`. If this value is `nil`, syntactic fontification uses the buffer's syntax table (the value returned by the function `syntax-table`; see ⟨undefined⟩ [Syntax Table Functions], page ⟨undefined⟩).

`font-lock-beginning-of-syntax-function` [Variable]
: If this variable is non-`nil`, it should be a function to move point back to a position that is syntactically at "top level" and outside of strings or comments. The value is normally set through an *other-vars* element in `font-lock-defaults`. If it is `nil`, Font Lock uses `syntax-begin-function` to move back outside of any comment, string, or sexp (see ⟨undefined⟩ [Position Parse], page ⟨undefined⟩).

  This variable is semi-obsolete; we usually recommend setting `syntax-begin-function` instead. One of its uses is to tune the behavior of syntactic fontification, e.g., to ensure that different kinds of strings or comments are highlighted differently.

  The specified function is called with no arguments. It should leave point at the beginning of any enclosing syntactic block. Typical values are `beginning-of-line` (used when the start of the line is known to be outside a syntactic block), or `beginning-of-defun` for programming modes, or `backward-paragraph` for textual modes.

`font-lock-syntactic-face-function` [Variable]
: If this variable is non-`nil`, it should be a function to determine which face to use for a given syntactic element (a string or a comment). The value is normally set through an *other-vars* element in `font-lock-defaults`.

  The function is called with one argument, the parse state at point returned by `parse-partial-sexp`, and should return a face. The default value returns `font-lock-comment-face` for comments and `font-lock-string-face` for strings (see Section 20.6.7 [Faces for Font Lock], page 436).

### 20.6.9 Multiline Font Lock Constructs

Normally, elements of `font-lock-keywords` should not match across multiple lines; that doesn't work reliably, because Font Lock usually scans just part of the buffer, and it can miss a multi-line construct that crosses the line boundary where the scan starts. (The scan normally starts at the beginning of a line.)

Making elements that match multiline constructs work properly has two aspects: correct *identification* and correct *rehighlighting*. The first means that Font Lock finds all multiline constructs. The second means that Font Lock will correctly rehighlight all the relevant text when a multiline construct is changed—for example, if some of the text that was previously part of a multiline construct ceases to be part of it. The two aspects are closely related, and often getting one of them to work will appear to make the other also work. However, for reliable results you must attend explicitly to both aspects.

There are three ways to ensure correct identification of multiline constructs:

- Add a function to `font-lock-extend-region-functions` that does the *identification* and extends the scan so that the scanned text never starts or ends in the middle of a multiline construct.
- Use the `font-lock-fontify-region-function` hook similarly to extend the scan so that the scanned text never starts or ends in the middle of a multiline construct.
- Somehow identify the multiline construct right when it gets inserted into the buffer (or at any point after that but before font-lock tries to highlight it), and mark it with a `font-lock-multiline` which will instruct font-lock not to start or end the scan in the middle of the construct.

There are three ways to do rehighlighting of multiline constructs:

- Place a `font-lock-multiline` property on the construct. This will rehighlight the whole construct if any part of it is changed. In some cases you can do this automatically by setting the `font-lock-multiline` variable, which see.
- Make sure `jit-lock-contextually` is set and rely on it doing its job. This will only rehighlight the part of the construct that follows the actual change, and will do it after a short delay. This only works if the highlighting of the various parts of your multiline construct never depends on text in subsequent lines. Since `jit-lock-contextually` is activated by default, this can be an attractive solution.
- Place a `jit-lock-defer-multiline` property on the construct. This works only if `jit-lock-contextually` is used, and with the same delay before rehighlighting, but like `font-lock-multiline`, it also handles the case where highlighting depends on subsequent lines.

#### 20.6.9.1 Font Lock Multiline

One way to ensure reliable rehighlighting of multiline Font Lock constructs is to put on them the text property `font-lock-multiline`. It should be present and non-`nil` for text that is part of a multiline construct.

When Font Lock is about to highlight a range of text, it first extends the boundaries of the range as necessary so that they do not fall within text marked with the `font-lock-multiline` property. Then it removes any `font-lock-multiline` properties from the range, and highlights it. The highlighting specification (mostly `font-lock-keywords`) must reinstall this property each time, whenever it is appropriate.

**Warning:** don't use the `font-lock-multiline` property on large ranges of text, because that will make rehighlighting slow.

`font-lock-multiline` [Variable]

If the `font-lock-multiline` variable is set to `t`, Font Lock will try to add the `font-lock-multiline` property automatically on multiline constructs. This is not a universal solution, however, since it slows down Font Lock somewhat. It can miss some multiline constructs, or make the property larger or smaller than necessary.

For elements whose *matcher* is a function, the function should ensure that submatch 0 covers the whole relevant multiline construct, even if only a small subpart will be highlighted. It is often just as easy to add the `font-lock-multiline` property by hand.

The `font-lock-multiline` property is meant to ensure proper refontification; it does not automatically identify new multiline constructs. Identifying the requires that Font Lock mode operate on large enough chunks at a time. This will happen by accident on many cases, which may give the impression that multiline constructs magically work. If you set the `font-lock-multiline` variable non-`nil`, this impression will be even stronger, since the highlighting of those constructs which are found will be properly updated from then on. But that does not work reliably.

To find multiline constructs reliably, you must either manually place the `font-lock-multiline` property on the text before Font Lock mode looks at it, or use `font-lock-fontify-region-function`.

#### 20.6.9.2 Region to Fontify after a Buffer Change

When a buffer is changed, the region that Font Lock refontifies is by default the smallest sequence of whole lines that spans the change. While this works well most of the time, sometimes it doesn't—for example, when a change alters the syntactic meaning of text on an earlier line.

You can enlarge (or even reduce) the region to refontify by setting the following variable:

`font-lock-extend-after-change-region-function` [Variable]

This buffer-local variable is either `nil` or a function for Font Lock mode to call to determine the region to scan and fontify.

The function is given three parameters, the standard *beg*, *end*, and *old-len* from `after-change-functions` (see Section 22.27 [Change Hooks], page 512). It should return either a cons of the beginning and end buffer positions (in that order) of the region to fontify, or `nil` (which means choose the region in the standard way). This function needs to preserve point, the match-data, and the current restriction. The region it returns may start or end in the middle of a line.

Since this function is called after every buffer change, it should be reasonably fast.

## 20.7 Automatic Indentation of code

For programming languages, an important feature of a major mode is to provide automatic indentation. This is controlled in Emacs by `indent-line-function` (see Section 22.17.2 [Mode-Specific Indent], page 484). Writing a good indentation function can be difficult and to a large extent it is still a black art.

Many major mode authors will start by writing a simple indentation function that works for simple cases, for example by comparing with the indentation of the previous text line. For most programming languages that are not really line-based, this tends to scale very poorly: improving such a function to let it handle more diverse situations tends to become more and more difficult, resulting in the end with a large, complex, unmaintainable indentation function which nobody dares to touch.

A good indentation function will usually need to actually parse the text, according to the syntax of the language. Luckily, it is not necessary to parse the text in as much detail as would be needed for a compiler, but on the other hand, the parser embedded in the indentation code will want to be somewhat friendly to syntactically incorrect code.

Good maintainable indentation functions usually fall into two categories: either parsing forward from some "safe" starting point until the position of interest, or parsing backward from the position of interest. Neither of the two is a clearly better choice than the other: parsing backward is often more difficult than parsing forward because programming languages are designed to be parsed forward, but for the purpose of indentation it has the advantage of not needing to guess a "safe" starting point, and it generally enjoys the property that only a minimum of text will be analyzed to decide the indentation of a line, so indentation will tend to be unaffected by syntax errors in some earlier unrelated piece of code. Parsing forward on the other hand is usually easier and has the advantage of making it possible to reindent efficiently a whole region at a time, with a single parse.

Rather than write your own indentation function from scratch, it is often preferable to try and reuse some existing ones or to rely on a generic indentation engine. There are sadly few such engines. The CC-mode indentation code (used with C, C++, Java, Awk and a few other such modes) has been made more generic over the years, so if your language seems somewhat similar to one of those languages, you might try to use that engine. Another one is SMIE which takes an approach in the spirit of Lisp sexps and adapts it to non-Lisp languages.

### 20.7.1 Simple Minded Indentation Engine

SMIE is a package that provides a generic navigation and indentation engine. Based on a very simple parser using an "operator precedence grammar", it lets major modes extend the sexp-based navigation of Lisp to non-Lisp languages as well as provide a simple to use but reliable auto-indentation.

Operator precedence grammar is a very primitive technology for parsing compared to some of the more common techniques used in compilers. It has the following characteristics: its parsing power is very limited, and it is largely unable to detect syntax errors, but it has the advantage of being algorithmically efficient and able to parse forward just as well as backward. In practice that means that SMIE can use it for indentation based on backward parsing, that it can provide both `forward-sexp` and `backward-sexp` functionality, and that it will naturally work on syntactically incorrect code without any extra effort. The downside is that it also means that most programming languages cannot be parsed correctly using SMIE, at least not without resorting to some special tricks (see Section 20.7.1.5 [SMIE Tricks], page 445).

#### 20.7.1.1 SMIE Setup and Features

SMIE is meant to be a one-stop shop for structural navigation and various other features which rely on the syntactic structure of code, in particular automatic indentation. The main entry point is `smie-setup` which is a function typically called while setting up a major mode.

`smie-setup` *grammar rules-function* **&rest** *keywords* [Function]

Setup SMIE navigation and indentation. *grammar* is a grammar table generated by `smie-prec2->grammar`. *rules-function* is a set of indentation rules for use on `smie-rules-function`. *keywords* are additional arguments, which can include the following keywords:

- `:forward-token` *fun*: Specify the forward lexer to use.

- `:backward-token` *fun*: Specify the backward lexer to use.

Calling this function is sufficient to make commands such as `forward-sexp`, `backward-sexp`, and `transpose-sexps` be able to properly handle structural elements other than just the paired parentheses already handled by syntax tables. For example, if the provided grammar is precise enough, `transpose-sexps` can correctly transpose the two arguments of a `+` operator, taking into account the precedence rules of the language.

Calling 'smie-setup' is also sufficient to make TAB indentation work in the expected way, extends `blink-matching-paren` to apply to elements like `begin...end`, and provides some commands that you can bind in the major mode keymap.

`smie-close-block` [Command]
This command closes the most recently opened (and not yet closed) block.

`smie-down-list` **&optional** *arg* [Command]
This command is like `down-list` but it also pays attention to nesting of tokens other than parentheses, such as `begin...end`.

### 20.7.1.2 Operator Precedence Grammars

SMIE's precedence grammars simply give to each token a pair of precedences: the left-precedence and the right-precedence. We say `T1 < T2` if the right-precedence of token `T1` is less than the left-precedence of token `T2`. A good way to read this `<` is as a kind of parenthesis: if we find `... T1 something T2 ...` then that should be parsed as `... T1 (something T2 ...` rather than as `... T1 something) T2 ...`. The latter interpretation would be the case if we had `T1 > T2`. If we have `T1 = T2`, it means that token T2 follows token T1 in the same syntactic construction, so typically we have `"begin" = "end"`. Such pairs of precedences are sufficient to express left-associativity or right-associativity of infix operators, nesting of tokens like parentheses and many other cases.

`smie-prec2->grammar` *table* [Function]
This function takes a *prec2* grammar *table* and returns an alist suitable for use in `smie-setup`. The *prec2 table* is itself meant to be built by one of the functions below.

`smie-merge-prec2s` **&rest** *tables* [Function]
This function takes several *prec2 tables* and merges them into a new *prec2* table.

`smie-precs->prec2` *precs* [Function]
This function builds a *prec2* table from a table of precedences *precs*. *precs* should be a list, sorted by precedence (for example `"+"` will come before `"*"`), of elements of the form `(assoc op ...)`, where each *op* is a token that acts as an operator; *assoc* is their associativity, which can be either `left`, `right`, `assoc`, or `nonassoc`. All operators in a given element share the same precedence level and associativity.

`smie-bnf->prec2` *bnf* **&rest** *resolvers* [Function]
This function lets you specify the grammar using a BNF notation. It accepts a *bnf* description of the grammar along with a set of conflict resolution rules *resolvers*, and returns a *prec2* table.

*bnf* is a list of nonterminal definitions of the form `(nonterm rhs1 rhs2 ...)` where each *rhs* is a (non-empty) list of terminals (aka tokens) or non-terminals.

Not all grammars are accepted:

- An *rhs* cannot be an empty list (an empty list is never needed, since SMIE allows all non-terminals to match the empty string anyway).
- An *rhs* cannot have 2 consecutive non-terminals: each pair of non-terminals needs to be separated by a terminal (aka token). This is a fundamental limitation of operator precedence grammars.

Additionally, conflicts can occur:

- The returned *prec2* table holds constraints between pairs of tokens, and for any given pair only one constraint can be present: T1 < T2, T1 = T2, or T1 > T2.
- A token can be an `opener` (something similar to an open-paren), a `closer` (like a close-paren), or `neither` of the two (e.g., an infix operator, or an inner token like `"else"`).

Precedence conflicts can be resolved via *resolvers*, which is a list of *precs* tables (see `smie-precs->prec2`): for each precedence conflict, if those `precs` tables specify a particular constraint, then the conflict is resolved by using this constraint instead, else a conflict is reported and one of the conflicting constraints is picked arbitrarily and the others are simply ignored.

### 20.7.1.3 Defining the Grammar of a Language

The usual way to define the SMIE grammar of a language is by defining a new global variable that holds the precedence table by giving a set of BNF rules. For example, the grammar definition for a small Pascal-like language could look like:

```
(require 'smie)
(defvar sample-smie-grammar
  (smie-prec2->grammar
   (smie-bnf->prec2
    '((id)
      (inst ("begin" insts "end")
            ("if" exp "then" inst "else" inst)
            (id ":=" exp)
            (exp))
      (insts (insts ";" insts) (inst))
      (exp (exp "+" exp)
           (exp "*" exp)
           ("(" exps ")"))
      (exps (exps "," exps) (exp)))
    '((assoc ";"))
    '((assoc ","))
    '((assoc "+") (assoc "*")))))
```

A few things to note:

- The above grammar does not explicitly mention the syntax of function calls: SMIE will automatically allow any sequence of sexps, such as identifiers, balanced parentheses, or `begin ... end` blocks to appear anywhere anyway.

- The grammar category `id` has no right hand side: this does not mean that it can match only the empty string, since as mentioned any sequence of sexps can appear anywhere anyway.
- Because non terminals cannot appear consecutively in the BNF grammar, it is difficult to correctly handle tokens that act as terminators, so the above grammar treats `";"` as a statement *separator* instead, which SMIE can handle very well.
- Separators used in sequences (such as `","` and `";"` above) are best defined with BNF rules such as `(foo (foo "separator" foo) ...)` which generate precedence conflicts which are then resolved by giving them an explicit `(assoc "separator")`.
- The `("(" exps ")")` rule was not needed to pair up parens, since SMIE will pair up any characters that are marked as having paren syntax in the syntax table. What this rule does instead (together with the definition of `exps`) is to make it clear that `","` should not appear outside of parentheses.
- Rather than have a single *precs* table to resolve conflicts, it is preferable to have several tables, so as to let the BNF part of the grammar specify relative precedences where possible.
- Unless there is a very good reason to prefer `left` or `right`, it is usually preferable to mark operators as associative, using `assoc`. For that reason `"+"` and `"*"` are defined above as `assoc`, although the language defines them formally as left associative.

#### 20.7.1.4 Defining Tokens

SMIE comes with a predefined lexical analyzer which uses syntax tables in the following way: any sequence of characters that have word or symbol syntax is considered a token, and so is any sequence of characters that have punctuation syntax. This default lexer is often a good starting point but is rarely actually correct for any given language. For example, it will consider `"2,+3"` to be composed of 3 tokens: `"2"`, `",+"`, and `"3"`.

To describe the lexing rules of your language to SMIE, you need 2 functions, one to fetch the next token, and another to fetch the previous token. Those functions will usually first skip whitespace and comments and then look at the next chunk of text to see if it is a special token. If so it should skip the token and return a description of this token. Usually this is simply the string extracted from the buffer, but it can be anything you want. For example:

```
(defvar sample-keywords-regexp
  (regexp-opt '("+" "*" "," ";" ">" ">=" "<" "<=" ":=" "=")))
(defun sample-smie-forward-token ()
  (forward-comment (point-max))
  (cond
   ((looking-at sample-keywords-regexp)
    (goto-char (match-end 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-forward "w_")
              (point))))))
```

```
(defun sample-smie-backward-token ()
  (forward-comment (- (point)))
  (cond
   ((looking-back sample-keywords-regexp (- (point) 2) t)
    (goto-char (match-beginning 0))
    (match-string-no-properties 0))
   (t (buffer-substring-no-properties
       (point)
       (progn (skip-syntax-backward "w_")
              (point))))))
```

Notice how those lexers return the empty string when in front of parentheses. This is because SMIE automatically takes care of the parentheses defined in the syntax table. More specifically if the lexer returns nil or an empty string, SMIE tries to handle the corresponding text as a sexp according to syntax tables.

#### 20.7.1.5 Living With a Weak Parser

The parsing technique used by SMIE does not allow tokens to behave differently in different contexts. For most programming languages, this manifests itself by precedence conflicts when converting the BNF grammar.

Sometimes, those conflicts can be worked around by expressing the grammar slightly differently. For example, for Modula-2 it might seem natural to have a BNF grammar that looks like this:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ...)
(cases (cases "|" cases)
       (caselabel ":" insts)
       ("ELSE" insts))
...
```

But this will create conflicts for `"ELSE"`: on the one hand, the IF rule implies (among many other things) that `"ELSE" = "END"`; but on the other hand, since `"ELSE"` appears within `cases`, which appears left of `"END"`, we also have `"ELSE" > "END"`. We can solve the conflict either by using:

```
...
(inst ("IF" exp "THEN" insts "ELSE" insts "END")
      ("CASE" exp "OF" cases "END")
      ("CASE" exp "OF" cases "ELSE" insts "END")
      ...)
(cases (cases "|" cases) (caselabel ":" insts))
...
```

or

```
...
(inst ("IF" exp "THEN" else "END")
      ("CASE" exp "OF" cases "END")
```

```
      ...)
(else (insts "ELSE" insts))
(cases (cases "|" cases) (caselabel ":" insts) (else))
...
```

Reworking the grammar to try and solve conflicts has its downsides, tho, because SMIE assumes that the grammar reflects the logical structure of the code, so it is preferable to keep the BNF closer to the intended abstract syntax tree.

Other times, after careful consideration you may conclude that those conflicts are not serious and simply resolve them via the *resolvers* argument of `smie-bnf->prec2`. Usually this is because the grammar is simply ambiguous: the conflict does not affect the set of programs described by the grammar, but only the way those programs are parsed. This is typically the case for separators and associative infix operators, where you want to add a resolver like `'((assoc "|"))`. Another case where this can happen is for the classic *dangling else* problem, where you will use `'((assoc "else" "then"))`. It can also happen for cases where the conflict is real and cannot really be resolved, but it is unlikely to pose a problem in practice.

Finally, in many cases some conflicts will remain despite all efforts to restructure the grammar. Do not despair: while the parser cannot be made more clever, you can make the lexer as smart as you want. So, the solution is then to look at the tokens involved in the conflict and to split one of those tokens into 2 (or more) different tokens. E.g., if the grammar needs to distinguish between two incompatible uses of the token `"begin"`, make the lexer return different tokens (say `"begin-fun"` and `"begin-plain"`) depending on which kind of `"begin"` it finds. This pushes the work of distinguishing the different cases to the lexer, which will thus have to look at the surrounding text to find ad-hoc clues.

#### 20.7.1.6 Specifying Indentation Rules

Based on the provided grammar, SMIE will be able to provide automatic indentation without any extra effort. But in practice, this default indentation style will probably not be good enough. You will want to tweak it in many different cases.

SMIE indentation is based on the idea that indentation rules should be as local as possible. To this end, it relies on the idea of *virtual* indentation, which is the indentation that a particular program point would have if it were at the beginning of a line. Of course, if that program point is indeed at the beginning of a line, its virtual indentation is its current indentation. But if not, then SMIE uses the indentation algorithm to compute the virtual indentation of that point. Now in practice, the virtual indentation of a program point does not have to be identical to the indentation it would have if we inserted a newline before it. To see how this works, the SMIE rule for indentation after a `{` in C does not care whether the `{` is standing on a line of its own or is at the end of the preceding line. Instead, these different cases are handled in the indentation rule that decides how to indent before a `{`.

Another important concept is the notion of *parent*: The *parent* of a token, is the head token of the nearest enclosing syntactic construct. For example, the parent of an `else` is the `if` to which it belongs, and the parent of an `if`, in turn, is the lead token of the surrounding construct. The command `backward-sexp` jumps from a token to its parent, but there are some caveats: for *openers* (tokens which start a construct, like `if`), you need to start with point before the token, while for others you need to start with point after the

token. `backward-sexp` stops with point before the parent token if that is the *opener* of the token of interest, and otherwise it stops with point after the parent token.

SMIE indentation rules are specified using a function that takes two arguments *method* and *arg* where the meaning of *arg* and the expected return value depend on *method*.

*method* can be:

- `:after`, in which case *arg* is a token and the function should return the *offset* to use for indentation after *arg*.
- `:before`, in which case *arg* is a token and the function should return the *offset* to use to indent *arg* itself.
- `:elem`, in which case the function should return either the offset to use to indent function arguments (if *arg* is the symbol `arg`) or the basic indentation step (if *arg* is the symbol `basic`).
- `:list-intro`, in which case *arg* is a token and the function should return non-`nil` if the token is followed by a list of expressions (not separated by any token) rather than an expression.

When *arg* is a token, the function is called with point just before that token. A return value of nil always means to fallback on the default behavior, so the function should return nil for arguments it does not expect.

*offset* can be:

- `nil`: use the default indentation rule.
- `(column . column)`: indent to column *column*.
- *number*: offset by *number*, relative to a base token which is the current token for `:after` and its parent for `:before`.

### 20.7.1.7 Helper Functions for Indentation Rules

SMIE provides various functions designed specifically for use in the indentation rules function (several of those functions break if used in another context). These functions all start with the prefix `smie-rule-`.

`smie-rule-bolp` [Function]
Return non-`nil` if the current token is the first on the line.

`smie-rule-hanging-p` [Function]
Return non-`nil` if the current token is *hanging*. A token is *hanging* if it is the last token on the line and if it is preceded by other tokens: a lone token on a line is not hanging.

`smie-rule-next-p` **&rest** *tokens* [Function]
Return non-`nil` if the next token is among *tokens*.

`smie-rule-prev-p` **&rest** *tokens* [Function]
Return non-`nil` if the previous token is among *tokens*.

`smie-rule-parent-p` **&rest** *parents* [Function]
Return non-`nil` if the current token's parent is among *parents*.

`smie-rule-sibling-p` [Function]
Return non-`nil` if the current token's parent is actually a sibling. This is the case for example when the parent of a `","` is just the previous `","`.

`smie-rule-parent` **&optional** *offset* [Function]
Return the proper offset to align the current token with the parent. If non-`nil`, *offset* should be an integer giving an additional offset to apply.

`smie-rule-separator` *method* [Function]
Indent current token as a *separator*.

By *separator*, we mean here a token whose sole purpose is to separate various elements within some enclosing syntactic construct, and which does not have any semantic significance in itself (i.e., it would typically not exist as a node in an abstract syntax tree).

Such a token is expected to have an associative syntax and be closely tied to its syntactic parent. Typical examples are `","` in lists of arguments (enclosed inside parentheses), or `";"` in sequences of instructions (enclosed in a `{...}` or `begin...end` block).

*method* should be the method name that was passed to 'smie-rules-function'.

### 20.7.1.8 Sample Indentation Rules

Here is an example of an indentation function:

```
(defun sample-smie-rules (kind token)
  (pcase (cons kind token)
    (`(:elem . basic) sample-indent-basic)
    (`(,_ . ",") (smie-rule-separator kind))
    (`(:after . ":=") sample-indent-basic)
    (`(:before . ,(or `"begin" `"(" `"{"))
     (if (smie-rule-hanging-p) (smie-rule-parent)))
    (`(:before . "if")
     (and (not (smie-rule-bolp)) (smie-rule-prev-p "else")
          (smie-rule-parent)))))
```

A few things to note:

- The first case indicates the basic indentation increment to use. If `sample-indent-basic` is nil, then SMIE uses the global setting `smie-indent-basic`. The major mode could have set `smie-indent-basic` buffer-locally instead, but that is discouraged.
- The rule for the token `","` make SMIE try to be more clever when the comma separator is placed at the beginning of lines. It tries to outdent the separator so as to align the code after the comma; for example:

```
x = longfunctionname (
        arg1
      , arg2
    );
```

- The rule for indentation after `":="` exists because otherwise SMIE would treat `":="` as an infix operator and would align the right argument with the left one.

- The rule for indentation before `"begin"` is an example of the use of virtual indentation: This rule is used only when `"begin"` is hanging, which can happen only when `"begin"` is not at the beginning of a line. So this is not used when indenting `"begin"` itself but only when indenting something relative to this `"begin"`. Concretely, this rule changes the indentation from:

```
if x > 0 then begin
        dosomething(x);
    end
```

  to

```
if x > 0 then begin
    dosomething(x);
end
```

- The rule for indentation before `"if"` is similar to the one for `"begin"`, but where the purpose is to treat `"else if"` as a single unit, so as to align a sequence of tests rather than indent each test further to the right. This function does this only in the case where the `"if"` is not placed on a separate line, hence the `smie-rule-bolp` test.

  If we know that the `"else"` is always aligned with its `"if"` and is always at the beginning of a line, we can use a more efficient rule:

```
((equal token "if")
 (and (not (smie-rule-bolp))
      (smie-rule-prev-p "else")
      (save-excursion
        (sample-smie-backward-token)
        (cons 'column (current-column)))))
```

  The advantage of this formulation is that it reuses the indentation of the previous `"else"`, rather than going all the way back to the first `"if"` of the sequence.

## 20.8 Desktop Save Mode

*Desktop Save Mode* is a feature to save the state of Emacs from one session to another. The user-level commands for using Desktop Save Mode are described in the GNU Emacs Manual (see Section "Saving Emacs Sessions" in *the GNU Emacs Manual*). Modes whose buffers visit a file, don't have to do anything to use this feature.

For buffers not visiting a file to have their state saved, the major mode must bind the buffer local variable `desktop-save-buffer` to a non-`nil` value.

`desktop-save-buffer` [Variable]

If this buffer-local variable is non-`nil`, the buffer will have its state saved in the desktop file at desktop save. If the value is a function, it is called at desktop save with argument *desktop-dirname*, and its value is saved in the desktop file along with the state of the buffer for which it was called. When file names are returned as part of the auxiliary information, they should be formatted using the call

```
(desktop-file-name file-name desktop-dirname)
```

For buffers not visiting a file to be restored, the major mode must define a function to do the job, and that function must be listed in the alist `desktop-buffer-mode-handlers`.

`desktop-buffer-mode-handlers` [Variable]

Alist with elements

```
(major-mode . restore-buffer-function)
```

The function *restore-buffer-function* will be called with argument list

```
(buffer-file-name buffer-name desktop-buffer-misc)
```

and it should return the restored buffer. Here *desktop-buffer-misc* is the value returned by the function optionally bound to `desktop-save-buffer`.

# 21 Indentation

*Indentation* refers to inserting or adjusting *whitespace characters* (space and/or tab characters) at the beginning of a line of text. This chapter documents indentation commands and options which are common to Text mode and related modes, as well as programming language modes. See Section 23.3 [Program Indent], page 517, for additional documentation about indenting in programming modes.

The simplest way to perform indentation is the TAB key. In most major modes, this runs the command `indent-for-tab-command`. (In C and related modes, TAB runs the command `c-indent-line-or-region`, which behaves similarly).

TAB Insert whitespace, or indent the current line, in a mode-appropriate way (`indent-for-tab-command`). If the region is active, indent all the lines within it.

The exact behavior of TAB depends on the major mode. In Text mode and related major modes, TAB normally inserts some combination of space and tab characters to advance point to the next tab stop (see Section 21.2 [Tab Stops], page 452). For this purpose, the position of the first non-whitespace character on the preceding line is treated as an additional tab stop, so you can use TAB to "align" point with the preceding line. If the region is active (see Section 8.3 [Using Region], page 91), TAB acts specially: it indents each line in the region so that its first non-whitespace character is aligned with the preceding line.

In programming modes, TAB indents the current line of code in a way that makes sense given the code in the preceding lines. If the region is active, all the lines in the region are indented this way. If point was initially within the current line's indentation, it is repositioned to the first non-whitespace character on the line.

If you just want to insert a tab character in the buffer, type `C-q TAB` (see Section 4.1 [Inserting Text], page 58).

## 21.1 Indentation Commands

Apart from the TAB (`indent-for-tab-command`) command, Emacs provides a variety of commands to perform indentation in other ways.

`C-j` Perform RET followed by TAB (`newline-and-indent`).

`C-M-o` Split the current line at point (`split-line`). The text on the line after point becomes a new line, indented to the same column where point is located. This command first moves point forward over any spaces and tabs. Afterward, point is positioned before the inserted newline.

`M-m` Move (forward or back) to the first non-whitespace character on the current line (`back-to-indentation`). If there are no non-whitespace characters on the line, move to the end of the line.

`M-i` Indent whitespace at point, up to the next tab stop (`tab-to-tab-stop`). See Section 21.2 [Tab Stops], page 452.

`M-x indent-relative`
Insert whitespace at point, until point is aligned with the first non-whitespace character on the previous line (actually, the last non-blank line). If point is

already farther right than that, run `tab-to-tab-stop` instead—unless called with a numeric argument, in which case do nothing.

`M-^` Merge the previous and the current line (`delete-indentation`). This "joins" the two lines cleanly, by replacing any indentation at the front of the current line, together with the line boundary, with a single space.

As a special case (useful for Lisp code), the single space is omitted if the characters to be joined are consecutive opening and closing parentheses, or if the junction follows another newline.

If there is a fill prefix, `M-^` deletes the fill prefix if it appears after the newline that is deleted. See ⟨undefined⟩ [Fill Prefix], page ⟨undefined⟩.

`C-M-\` Indent all the lines in the region, as though you had typed TAB at the beginning of each line (`indent-region`).

If a numeric argument is supplied, indent every line in the region to that column number.

`C-x TAB` Shift each line in the region by a fixed distance, to the right or left (`indent-rigidly`). The distance to move is determined by the numeric argument (positive to move rightward, negative to move leftward).

This command can be used to remove all indentation from the lines in the region, by invoking it with a large negative argument, e.g., `C-u -1000 C-x TAB`.

## 21.2 Tab Stops

Emacs defines certain column numbers to be *tab stops*. These are used as stopping points by TAB when inserting whitespace in Text mode and related modes (see Section 22.17 [Indentation], page 483), and by commands like `M-i` (see Section 21.1 [Indentation Commands], page 451). By default, tab stops are located every 8 columns. These positions are stored in the variable `tab-stop-list`, whose value is a list of column numbers in increasing order.

Instead of customizing the variable `tab-stop-list` directly, a convenient way to view and set tab stops is via the command `M-x edit-tab-stops`. This switches to a buffer containing a description of the tab stop settings, which looks like this:

```
           :       :       :       :       :       :
0          1       2       3       4
0123456789012345678901234567890123456789012345678
To install changes, type C-c C-c
```

The first line contains a colon at each tab stop. The numbers on the next two lines are present just to indicate where the colons are.

You can edit this buffer to specify different tab stops by placing colons on the desired columns. The buffer uses Overwrite mode (see Section 20.3 [Minor Modes], page 413). When you are done, type `C-c C-c` to make the new tab stops take effect. Normally, the new tab stop settings apply to all buffers. However, if you have made the `tab-stop-list` variable local to the buffer where you called `M-x edit-tab-stops` (see Section 33.2.3 [Locals], page 697), then the new tab stop settings apply only to that buffer. To save the tab stop settings for future Emacs sessions, use the Customize interface to save the value of `tab-stop-list` (see Section 33.1 [Easy Customization], page 686).

Note that the tab stops discussed in this section have nothing to do with how tab characters are displayed in the buffer. Tab characters are always displayed as empty spaces extending to the next *display tab stop*. See ⟨undefined⟩ [Text Display], page ⟨undefined⟩.

## 21.3 Tabs vs. Spaces

Normally, indentation commands insert (or remove) an optimal mix of space characters and tab characters to align to the desired column. Tab characters are displayed as a stretch of empty space extending to the next *display tab stop*. By default, there is one display tab stop every `tab-width` columns (the default is 8). See ⟨undefined⟩ [Text Display], page ⟨undefined⟩.

If you prefer, all indentation can be made from spaces only. To request this, set the buffer-local variable `indent-tabs-mode` to `nil`. See Section 33.2.3 [Locals], page 697, for information about setting buffer-local variables. Note, however, that `C-q TAB` always inserts a tab character, regardless of the value of `indent-tabs-mode`.

One reason to set `indent-tabs-mode` to `nil` is that not all editors display tab characters in the same way. Emacs users, too, may have different customized values of `tab-width`. By using spaces only, you can make sure that your file always looks the same. If you only care about how it looks within Emacs, another way to tackle this problem is to set the `tab-width` variable in a file-local variable (see Section 33.2.4 [File Variables], page 698).

There are also commands to convert tabs to spaces or vice versa, always preserving the columns of all non-whitespace text. `M-x tabify` scans the region for sequences of spaces, and converts sequences of at least two spaces to tabs if that can be done without changing indentation. `M-x untabify` changes all tabs in the region to appropriate numbers of spaces.

## 21.4 Convenience Features for Indentation

The variable `tab-always-indent` tweaks the behavior of the TAB (`indent-for-tab-command`) command. The default value, `t`, gives the behavior described in Section 22.17 [Indentation], page 483. If you change the value to the symbol `complete`, then TAB first tries to indent the current line, and if the line was already indented, it tries to complete the text at point (see Section 23.8 [Symbol Completion], page 528). If the value is `nil`, then TAB indents the current line only if point is at the left margin or in the line's indentation; otherwise, it inserts a tab character.

Electric Indent mode is a global minor mode that automatically indents the line after every RET you type. To toggle this minor mode, type `M-x electric-indent-mode`.

# 22 Text

This chapter describes the functions that deal with the text in a buffer. Most examine, insert, or delete text in the current buffer, often operating at point or on text adjacent to point. Many are interactive. All the functions that change the text provide for undoing the changes (see Section 22.9 [Undo], page 469).

Many text-related functions operate on a region of text defined by two buffer positions passed in arguments named *start* and *end*. These arguments should be either markers (see ⟨undefined⟩ [Markers], page ⟨undefined⟩) or numeric character positions (see ⟨undefined⟩ [Positions], page ⟨undefined⟩). The order of these arguments does not matter; it is all right for *start* to be the end of the region and *end* the beginning. For example, `(delete-region 1 10)` and `(delete-region 10 1)` are equivalent. An `args-out-of-range` error is signaled if either *start* or *end* is outside the accessible portion of the buffer. In an interactive call, point and the mark are used for these arguments.

Throughout this chapter, "text" refers to the characters in the buffer, together with their properties (when relevant). Keep in mind that point is always between two characters, and the cursor appears on the character after point.

## 22.1 Examining Text Near Point

Many functions are provided to look at the characters around point. Several simple functions are described here. See also `looking-at` in Section 12.5 [Regexp Search], page 205.

In the following four functions, "beginning" or "end" of buffer refers to the beginning or end of the accessible portion.

`char-after` **&optional** *position* [Function]

This function returns the character in the current buffer at (i.e., immediately after) position *position*. If *position* is out of range for this purpose, either before the beginning of the buffer, or at or beyond the end, then the value is `nil`. The default for *position* is point.

In the following example, assume that the first character in the buffer is '`@`':

```
(string (char-after 1))
     ⇒ "@"
```

`char-before` **&optional** *position* [Function]

This function returns the character in the current buffer immediately before position *position*. If *position* is out of range for this purpose, either at or before the beginning of the buffer, or beyond the end, then the value is `nil`. The default for *position* is point.

`following-char` [Function]

This function returns the character following point in the current buffer. This is similar to `(char-after (point))`. However, if point is at the end of the buffer, then `following-char` returns 0.

Remember that point is always between characters, and the cursor normally appears over the character following point. Therefore, the character returned by `following-char` is the character the cursor is over.

In this example, point is between the '`a`' and the '`c`'.

```
---------- Buffer: foo ----------
Gentlemen may cry ‘‘Pea⋆ce! Peace!,’’
but there is no peace.
---------- Buffer: foo ----------

(string (preceding-char))
     ⇒ "a"
(string (following-char))
     ⇒ "c"
```

`preceding-char` [Function]
: This function returns the character preceding point in the current buffer. See above, under `following-char`, for an example. If point is at the beginning of the buffer, `preceding-char` returns 0.

`bobp` [Function]
: This function returns `t` if point is at the beginning of the buffer. If narrowing is in effect, this means the beginning of the accessible portion of the text. See also `point-min` in Section 1.1 [Point], page 6.

`eobp` [Function]
: This function returns `t` if point is at the end of the buffer. If narrowing is in effect, this means the end of accessible portion of the text. See also `point-max` in See Section 1.1 [Point], page 6.

`bolp` [Function]
: This function returns `t` if point is at the beginning of a line. See ⟨undefined⟩ [Text Lines], page ⟨undefined⟩. The beginning of the buffer (or of its accessible portion) always counts as the beginning of a line.

`eolp` [Function]
: This function returns `t` if point is at the end of a line. The end of the buffer (or of its accessible portion) is always considered the end of a line.

## 22.2 Examining Buffer Contents

This section describes functions that allow a Lisp program to convert any portion of the text in the buffer into a string.

`buffer-substring` *start end* [Function]
: This function returns a string containing a copy of the text of the region defined by positions *start* and *end* in the current buffer. If the arguments are not positions in the accessible portion of the buffer, `buffer-substring` signals an `args-out-of-range` error.

  Here's an example which assumes Font-Lock mode is not enabled:

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------
```

```
(buffer-substring 1 10)
     ⇒ "This is t"
(buffer-substring (point-max) 10)
     ⇒ "he contents of buffer foo\n"
```

If the text being copied has any text properties, these are copied into the string along with the characters they belong to. See Section 22.19 [Text Properties], page 489. However, overlays (see Section 11.9 [Overlays], page 128) in the buffer and their properties are ignored, not copied.

For example, if Font-Lock mode is enabled, you might get results like these:

```
(buffer-substring 1 10)
     ⇒ #("This is t" 0 1 (fontified t) 1 9 (fontified t))
```

`buffer-substring-no-properties` *start end* [Function]

This is like `buffer-substring`, except that it does not copy text properties, just the characters themselves. See Section 22.19 [Text Properties], page 489.

`buffer-string` [Function]

This function returns the contents of the entire accessible portion of the current buffer, as a string.

`filter-buffer-substring` *start end* **&optional** *delete* [Function]

This function passes the buffer text between *start* and *end* through the filter functions specified by the wrapper hook `filter-buffer-substring-functions`, and returns the result. The obsolete variable `buffer-substring-filters` is also consulted. If both of these variables are `nil`, the value is the unaltered text from the buffer, i.e., what `buffer-substring` would return.

If *delete* is non-`nil`, this function deletes the text between *start* and *end* after copying it, like `delete-and-extract-region`.

Lisp code should use this function instead of `buffer-substring`, `buffer-substring-no-properties`, or `delete-and-extract-region` when copying into user-accessible data structures such as the kill-ring, X clipboard, and registers. Major and minor modes can add functions to `filter-buffer-substring-functions` to alter such text as it is copied out of the buffer.

`filter-buffer-substring-functions` [Variable]

This variable is a wrapper hook (see Section 20.1.1 [Running Hooks], page 396), whose members should be functions that accept four arguments: *fun*, *start*, *end*, and *delete*. *fun* is a function that takes three arguments (*start*, *end*, and *delete*), and returns a string. In both cases, the *start*, *end*, and *delete* arguments are the same as those of `filter-buffer-substring`.

The first hook function is passed a *fun* that is equivalent to the default operation of `filter-buffer-substring`, i.e., it returns the buffer-substring between *start* and *end* (processed by any `buffer-substring-filters`) and optionally deletes the original text from the buffer. In most cases, the hook function will call *fun* once, and then do its own processing of the result. The next hook function receives a *fun* equivalent to this, and so on. The actual return value is the result of all the hook functions acting in sequence.

`buffer-substring-filters` [Variable]

This variable is obsoleted by `filter-buffer-substring-functions`, but is still supported for backward compatibility. Its value should should be a list of functions which accept a single string argument and return another string. `filter-buffer-substring` passes the buffer substring to the first function in this list, and the return value of each function is passed to the next function. The return value of the last function is passed to `filter-buffer-substring-functions`.

`current-word` **&optional** *strict really-word* [Function]

This function returns the symbol (or word) at or near point, as a string. The return value includes no text properties.

If the optional argument *really-word* is non-`nil`, it finds a word; otherwise, it finds a symbol (which includes both word characters and symbol constituent characters).

If the optional argument *strict* is non-`nil`, then point must be in or next to the symbol or word—if no symbol or word is there, the function returns `nil`. Otherwise, a nearby symbol or word on the same line is acceptable.

`thing-at-point` *thing* [Function]

Return the *thing* around or next to point, as a string.

The argument *thing* is a symbol which specifies a kind of syntactic entity. Possibilities include `symbol`, `list`, `sexp`, `defun`, `filename`, `url`, `word`, `sentence`, `whitespace`, `line`, `page`, and others.

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea*ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------

(thing-at-point 'word)
     ⇒ "Peace"
(thing-at-point 'line)
     ⇒ "Gentlemen may cry ``Peace! Peace!,''\n"
(thing-at-point 'whitespace)
     ⇒ nil
```

## 22.3 Comparing Text

This function lets you compare portions of the text in a buffer, without copying them into strings first.

`compare-buffer-substrings` *buffer1 start1 end1 buffer2 start2 end2* [Function]

This function lets you compare two substrings of the same buffer or two different buffers. The first three arguments specify one substring, giving a buffer (or a buffer name) and two positions within the buffer. The last three arguments specify the other substring in the same way. You can use `nil` for *buffer1*, *buffer2*, or both to stand for the current buffer.

The value is negative if the first substring is less, positive if the first is greater, and zero if they are equal. The absolute value of the result is one plus the index of the first differing characters within the substrings.

This function ignores case when comparing characters if `case-fold-search` is non-`nil`. It always ignores text properties.

Suppose the current buffer contains the text '`foobarbar haha!rara!`'; then in this example the two substrings are '`rbar `' and '`rara!`'. The value is 2 because the first substring is greater at the second character.

```
(compare-buffer-substrings nil 6 11 nil 16 21)
     ⇒ 2
```

## 22.4 Inserting Text

*Insertion* means adding new text to a buffer. The inserted text goes at point—between the character before point and the character after point. Some insertion functions leave point before the inserted text, while other functions leave it after. We call the former insertion *after point* and the latter insertion *before point*.

Insertion moves markers located at positions after the insertion point, so that they stay with the surrounding text (see ⟨undefined⟩ [Markers], page ⟨undefined⟩). When a marker points at the place of insertion, insertion may or may not relocate the marker, depending on the marker's insertion type (see ⟨undefined⟩ [Marker Insertion Types], page ⟨undefined⟩). Certain special functions such as `insert-before-markers` relocate all such markers to point after the inserted text, regardless of the markers' insertion type.

Insertion functions signal an error if the current buffer is read-only or if they insert within read-only text.

These functions copy text characters from strings and buffers along with their properties. The inserted characters have exactly the same properties as the characters they were copied from. By contrast, characters specified as separate arguments, not part of a string or buffer, inherit their text properties from the neighboring text.

The insertion functions convert text from unibyte to multibyte in order to insert in a multibyte buffer, and vice versa—if the text comes from a string or from a buffer. However, they do not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See ⟨undefined⟩ [Converting Representations], page ⟨undefined⟩.

`insert` **&rest** *args* [Function]

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. In other words, it inserts the text before point. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

`insert-before-markers` **&rest** *args* [Function]

This function inserts the strings and/or characters *args* into the current buffer, at point, moving point forward. An error is signaled unless all *args* are either strings or characters. The value is `nil`.

This function is unlike the other insertion functions in that it relocates markers initially pointing at the insertion point, to point after the inserted text. If an overlay begins at the insertion point, the inserted text falls outside the overlay; if a nonempty overlay ends at the insertion point, the inserted text falls inside that overlay.

`insert-char` *character* **&optional** *count inherit* [Command]
: This command inserts *count* instances of *character* into the current buffer before point. The argument *count* must be an integer, and *character* must be a character.

  If called interactively, this command prompts for *character* using its Unicode name or its code point. See Section "Inserting Text" in *The GNU Emacs Manual*.

  This function does not convert unibyte character codes 128 through 255 to multibyte characters, not even if the current buffer is a multibyte buffer. See ⟨undefined⟩ [Converting Representations], page ⟨undefined⟩.

  If *inherit* is non-`nil`, the inserted characters inherit sticky text properties from the two characters before and after the insertion point. See Section 22.19.6 [Sticky Properties], page 500.

`insert-buffer-substring` *from-buffer-or-name* **&optional** *start end* [Function]
: This function inserts a portion of buffer *from-buffer-or-name* (which must already exist) into the current buffer before point. The text inserted is the region between *start* and *end*. (These arguments default to the beginning and end of the accessible portion of that buffer.) This function returns `nil`.

  In this example, the form is executed with buffer '`bar`' as the current buffer. We assume that buffer '`bar`' is initially empty.

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth⋆
---------- Buffer: bar ----------
```

`insert-buffer-substring-no-properties` *from-buffer-or-name* [Function]
**&optional** *start end*
: This is like `insert-buffer-substring` except that it does not copy any text properties.

See Section 22.19.6 [Sticky Properties], page 500, for other insertion functions that inherit text properties from the nearby text in addition to inserting it. Whitespace inserted by indentation functions also inherits text properties.

## 22.5 User-Level Insertion Commands

This section describes higher-level commands for inserting text, commands intended primarily for the user but useful also in Lisp programs.

`insert-buffer` *from-buffer-or-name* [Command]
: This command inserts the entire accessible contents of *from-buffer-or-name* (which must exist) into the current buffer after point. It leaves the mark after the inserted text. The value is `nil`.

`self-insert-command` *count* [Command]
: This command inserts the last character typed; it does so *count* times, before point, and returns `nil`. Most printing characters are bound to this command. In routine use, `self-insert-command` is the most frequently called function in Emacs, but programs rarely use it except to install it on a keymap.

  In an interactive call, *count* is the numeric prefix argument.

  Self-insertion translates the input character through `translation-table-for-input`. See ⟨undefined⟩ [Translation of Characters], page ⟨undefined⟩.

  This command calls `auto-fill-function` whenever that is non-`nil` and the character inserted is in the table `auto-fill-chars` (see Section 22.14 [Auto Filling], page 478).

  This command performs abbrev expansion if Abbrev mode is enabled and the inserted character does not have word-constituent syntax. (See Chapter 26 [Abbrevs], page 581, and ⟨undefined⟩ [Syntax Class Table], page ⟨undefined⟩.) It is also responsible for calling `blink-paren-function` when the inserted character has close parenthesis syntax (see Section 11.19 [Blinking], page 188).

  The final thing this command does is to run the hook `post-self-insert-hook`. You could use this to automatically reindent text as it is typed, for example.

  Do not try substituting your own definition of `self-insert-command` for the standard one. The editor command loop handles this function specially.

`newline` **&optional** *number-of-newlines* [Command]
: This command inserts newlines into the current buffer before point. If *number-of-newlines* is supplied, that many newline characters are inserted.

  This function calls `auto-fill-function` if the current column number is greater than the value of `fill-column` and *number-of-newlines* is `nil`. Typically what `auto-fill-function` does is insert a newline; thus, the overall result in this case is to insert two newlines at different places: one at point, and another earlier in the line. `newline` does not auto-fill if *number-of-newlines* is non-`nil`.

  This command indents to the left margin if that is not zero. See Section 22.12 [Margins], page 475.

  The value returned is `nil`. In an interactive call, *count* is the numeric prefix argument.

`overwrite-mode` [Variable]
: This variable controls whether overwrite mode is in effect. The value should be `overwrite-mode-textual`, `overwrite-mode-binary`, or `nil`. `overwrite-mode-textual` specifies textual overwrite mode (treats newlines and tabs specially), and `overwrite-mode-binary` specifies binary overwrite mode (treats newlines and tabs like any other characters).

## 22.6 Deleting Text

Deletion means removing part of the text in a buffer, without saving it in the kill ring (see Section 22.8 [The Kill Ring], page 464). Deleted text can't be yanked, but can be reinserted using the undo mechanism (see Section 22.9 [Undo], page 469). Some deletion functions do save text in the kill ring in some special cases.

All of the deletion functions operate on the current buffer.

`erase-buffer` [Command]

This function deletes the entire text of the current buffer (*not* just the accessible portion), leaving it empty. If the buffer is read-only, it signals a `buffer-read-only` error; if some of the text in it is read-only, it signals a `text-read-only` error. Otherwise, it deletes the text without asking for any confirmation. It returns `nil`.

Normally, deleting a large amount of text from a buffer inhibits further auto-saving of that buffer "because it has shrunk". However, `erase-buffer` does not do this, the idea being that the future text is not really related to the former text, and its size should not be compared with that of the former text.

`delete-region` *start end* [Command]

This command deletes the text between positions *start* and *end* in the current buffer, and returns `nil`. If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

`delete-and-extract-region` *start end* [Function]

This function deletes the text between positions *start* and *end* in the current buffer, and returns a string containing the text just deleted.

If point was inside the deleted region, its value afterward is *start*. Otherwise, point relocates with the surrounding text, as markers do.

`delete-char` *count* **&optional** *killp* [Command]

This command deletes *count* characters directly after point, or before point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`delete-backward-char` *count* **&optional** *killp* [Command]

This command deletes *count* characters directly before point, or after point if *count* is negative. If *killp* is non-`nil`, then it saves the deleted characters in the kill ring.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`backward-delete-char-untabify` *count* **&optional** *killp* [Command]

This command deletes *count* characters backward, changing tabs into spaces. When the next character to be deleted is a tab, it is first replaced with the proper number of spaces to preserve alignment and then one of those spaces is deleted instead of the tab. If *killp* is non-`nil`, then the command saves the deleted characters in the kill ring.

Conversion of tabs to spaces happens only if *count* is positive. If it is negative, exactly $-$*count* characters after point are deleted.

In an interactive call, *count* is the numeric prefix argument, and *killp* is the unprocessed prefix argument. Therefore, if a prefix argument is supplied, the text is saved in the kill ring. If no prefix argument is supplied, then one character is deleted, but not saved in the kill ring.

The value returned is always `nil`.

`backward-delete-char-untabify-method` [User Option]

This option specifies how `backward-delete-char-untabify` should deal with whitespace. Possible values include `untabify`, the default, meaning convert a tab to many spaces and delete one; `hungry`, meaning delete all tabs and spaces before point with one command; `all` meaning delete all tabs, spaces and newlines before point, and `nil`, meaning do nothing special for whitespace characters.

## 22.7 User-Level Deletion Commands

This section describes higher-level commands for deleting text, commands intended primarily for the user but useful also in Lisp programs.

`delete-horizontal-space` **&optional** *backward-only* [Command]

This function deletes all spaces and tabs around point. It returns `nil`.

If *backward-only* is non-`nil`, the function deletes spaces and tabs before point, but not after point.

In the following examples, we call `delete-horizontal-space` four times, once on each line, with point between the second and third characters on the line each time.

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------

(delete-horizontal-space)   ; Four times.
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

`delete-indentation` **&optional** *join-following-p* [Command]

This function joins the line point is on to the previous line, deleting any whitespace at the join and in some cases replacing it with one space. If *join-following-p* is non-`nil`, `delete-indentation` joins this line to the following line instead. The function returns `nil`.

If there is a fill prefix, and the second of the lines being joined starts with the prefix, then `delete-indentation` deletes the fill prefix before joining the lines. See Section 22.12 [Margins], page 475.

In the example below, point is located on the line starting '`events`', and it makes no difference if there are trailing spaces in the preceding line.

```
---------- Buffer: foo ----------
When in the course of human
*    events, it becomes necessary
---------- Buffer: foo ----------

(delete-indentation)
     ⇒ nil

---------- Buffer: foo ----------
When in the course of human* events, it becomes necessary
---------- Buffer: foo ----------
```

After the lines are joined, the function `fixup-whitespace` is responsible for deciding whether to leave a space at the junction.

`fixup-whitespace` [Command]

This function replaces all the horizontal whitespace surrounding point with either one space or no space, according to the context. It returns `nil`.

At the beginning or end of a line, the appropriate amount of space is none. Before a character with close parenthesis syntax, or after a character with open parenthesis or expression-prefix syntax, no space is also appropriate. Otherwise, one space is appropriate. See ⟨undefined⟩ [Syntax Class Table], page ⟨undefined⟩.

In the example below, `fixup-whitespace` is called the first time with point before the word '`spaces`' in the first line. For the second invocation, point is directly after the '`(`'.

```
---------- Buffer: foo ----------
This has too many     *spaces
This has too many spaces at the start of (*   this list)
---------- Buffer: foo ----------

(fixup-whitespace)
     ⇒ nil
(fixup-whitespace)
     ⇒ nil

---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

`just-one-space` **&optional** *n* [Command]

This command replaces any spaces and tabs around point with a single space, or *n* spaces if *n* is specified. It returns `nil`.

`delete-blank-lines` [Command]

This function deletes blank lines surrounding point. If point is on a blank line with one or more blank lines before or after it, then all but one of them are deleted. If

point is on an isolated blank line, then it is deleted. If point is on a nonblank line, the command deletes all blank lines immediately following it.

A blank line is defined as a line containing only tabs and spaces.

`delete-blank-lines` returns `nil`.

## 22.8 The Kill Ring

*Kill functions* delete text like the deletion functions, but save it so that the user can reinsert it by *yanking*. Most of these functions have '`kill-`' in their name. By contrast, the functions whose names start with '`delete-`' normally do not save text for yanking (though they can still be undone); these are "deletion" functions.

Most of the kill commands are primarily for interactive use, and are not described here. What we do describe are the functions provided for use in writing such commands. You can use these functions to write commands for killing text. When you need to delete text for internal purposes within a Lisp function, you should normally use deletion functions, so as not to disturb the kill ring contents. See Section 22.6 [Deletion], page 460.

Killed text is saved for later yanking in the *kill ring*. This is a list that holds a number of recent kills, not just the last text kill. We call this a "ring" because yanking treats it as having elements in a cyclic order. The list is kept in the variable `kill-ring`, and can be operated on with the usual functions for lists; there are also specialized functions, described in this section, that treat it as a ring.

Some people think this use of the word "kill" is unfortunate, since it refers to operations that specifically *do not* destroy the entities "killed". This is in sharp contrast to ordinary life, in which death is permanent and "killed" entities do not come back to life. Therefore, other metaphors have been proposed. For example, the term "cut ring" makes sense to people who, in pre-computer days, used scissors and paste to cut up and rearrange manuscripts. However, it would be difficult to change the terminology now.

### 22.8.1 Kill Ring Concepts

The kill ring records killed text as strings in a list, most recent first. A short kill ring, for example, might look like this:

```
("some text" "a different piece of text" "even older text")
```

When the list reaches `kill-ring-max` entries in length, adding a new entry automatically deletes the last entry.

When kill commands are interwoven with other commands, each kill command makes a new entry in the kill ring. Multiple kill commands in succession build up a single kill ring entry, which would be yanked as a unit; the second and subsequent consecutive kill commands add text to the entry made by the first one.

For yanking, one entry in the kill ring is designated the "front" of the ring. Some yank commands "rotate" the ring by designating a different element as the "front". But this virtual rotation doesn't change the list itself—the most recent entry always comes first in the list.

### 22.8.2 Functions for Killing

`kill-region` is the usual subroutine for killing text. Any command that calls this function is a "kill command" (and should probably have '`kill`' in its name). `kill-region` puts the

newly killed text in a new element at the beginning of the kill ring or adds it to the most recent element. It determines automatically (using `last-command`) whether the previous command was a kill command, and if so appends the killed text to the most recent entry.

`kill-region` *start end* [Command]
: This function kills the text in the region defined by *start* and *end*. The text is deleted but saved in the kill ring, along with its text properties. The value is always `nil`.

  In an interactive call, *start* and *end* are point and the mark.

  If the buffer or text is read-only, `kill-region` modifies the kill ring just the same, then signals an error without modifying the buffer. This is convenient because it lets the user use a series of kill commands to copy text from a read-only buffer into the kill ring.

`kill-read-only-ok` [User Option]
: If this option is non-`nil`, `kill-region` does not signal an error if the buffer or text is read-only. Instead, it simply returns, updating the kill ring but not changing the buffer.

`copy-region-as-kill` *start end* [Command]
: This command saves the region defined by *start* and *end* on the kill ring (including text properties), but does not delete the text from the buffer. It returns `nil`.

  The command does not set `this-command` to `kill-region`, so a subsequent kill command does not append to the same kill ring entry.

  In Lisp programs, it is better to use `kill-new` or `kill-append` instead of this command. See Section 22.8.5 [Low-Level Kill Ring], page 467.

### 22.8.3 Yanking

Yanking means inserting text from the kill ring, but it does not insert the text blindly. The `yank` command, and related commands, use `insert-for-yank` to perform special processing on the text before it is inserted.

`insert-for-yank` *string* [Function]
: This function works like `insert`, except that it processes the text in *string* according to the `yank-handler` text property, as well as the variables `yank-handled-properties` and `yank-excluded-properties` (see below), before inserting the result into the current buffer.

`insert-buffer-substring-as-yank` *buf* **&optional** *start end* [Function]
: This function resembles `insert-buffer-substring`, except that it processes the text according to `yank-handled-properties` and `yank-excluded-properties`. (It does not handle the `yank-handler` property, which does not normally occur in buffer text anyway.)

If you put a `yank-handler` text property on all or part of a string, that alters how `insert-for-yank` inserts the string. If different parts of the string have different `yank-handler` values (comparison being done with `eq`), each substring is handled separately. The property value must be a list of one to four elements, with the following format (where elements after the first may be omitted):

`(function param noexclude undo)`

Here is what the elements do:

*function* When *function* is non-`nil`, it is called instead of `insert` to insert the string, with one argument—the string to insert.

*param* If *param* is present and non-`nil`, it replaces *string* (or the substring of *string* being processed) as the object passed to *function* (or `insert`). For example, if *function* is `yank-rectangle`, *param* should be a list of strings to insert as a rectangle.

*noexclude* If *noexclude* is present and non-`nil`, that disables the normal action of `yank-handled-properties` and `yank-excluded-properties` on the inserted string.

*undo* If *undo* is present and non-`nil`, it is a function that will be called by `yank-pop` to undo the insertion of the current object. It is called with two arguments, the start and end of the current region. *function* can set `yank-undo-function` to override the *undo* value.

`yank-handled-properties` [User Option]

This variable specifies special text property handling conditions for yanked text. It takes effect after the text has been inserted (either normally, or via the `yank-handler` property), and prior to `yank-excluded-properties` taking effect.

The value should be an alist of elements `(prop . fun)`. Each alist element is handled in order. The inserted text is scanned for stretches of text having text properties `eq` to *prop*; for each such stretch, *fun* is called with three arguments: the value of the property, and the start and end positions of the text.

`yank-excluded-properties` [User Option]

The value of this variable is the list of properties to remove from inserted text. Its default value contains properties that might lead to annoying results, such as causing the text to respond to the mouse or specifying key bindings. It takes effect after `yank-handled-properties`.

### 22.8.4 Functions for Yanking

This section describes higher-level commands for yanking, which are intended primarily for the user but useful also in Lisp programs. Both `yank` and `yank-pop` honor the `yank-excluded-properties` variable and `yank-handler` text property (see Section 22.8.3 [Yanking], page 465).

`yank` **&optional** *arg* [Command]

This command inserts before point the text at the front of the kill ring. It sets the mark at the beginning of that text, using `push-mark` (see ⟨undefined⟩ [The Mark], page ⟨undefined⟩), and puts point at the end.

If *arg* is a non-`nil` list (which occurs interactively when the user types `C-u` with no digits), then `yank` inserts the text as described above, but puts point before the yanked text and sets the mark after it.

If *arg* is a number, then `yank` inserts the *arg*th most recently killed text—the *arg*th element of the kill ring list, counted cyclically from the front, which is considered the first element for this purpose.

`yank` does not alter the contents of the kill ring, unless it used text provided by another program, in which case it pushes that text onto the kill ring. However if *arg* is an integer different from one, it rotates the kill ring to place the yanked string at the front.

`yank` returns `nil`.

`yank-pop` **&optional** *arg* [Command]

This command replaces the just-yanked entry from the kill ring with a different entry from the kill ring.

This is allowed only immediately after a `yank` or another `yank-pop`. At such a time, the region contains text that was just inserted by yanking. `yank-pop` deletes that text and inserts in its place a different piece of killed text. It does not add the deleted text to the kill ring, since it is already in the kill ring somewhere. It does however rotate the kill ring to place the newly yanked string at the front.

If *arg* is `nil`, then the replacement text is the previous element of the kill ring. If *arg* is numeric, the replacement is the *arg*th previous kill. If *arg* is negative, a more recent kill is the replacement.

The sequence of kills in the kill ring wraps around, so that after the oldest one comes the newest one, and before the newest one goes the oldest.

The return value is always `nil`.

`yank-undo-function` [Variable]

If this variable is non-`nil`, the function `yank-pop` uses its value instead of `delete-region` to delete the text inserted by the previous `yank` or `yank-pop` command. The value must be a function of two arguments, the start and end of the current region.

The function `insert-for-yank` automatically sets this variable according to the *undo* element of the `yank-handler` text property, if there is one.

### 22.8.5 Low-Level Kill Ring

These functions and variables provide access to the kill ring at a lower level, but are still convenient for use in Lisp programs, because they take care of interaction with window system selections (see Section 18.18 [Window System Selections], page 366).

`current-kill` *n* **&optional** *do-not-move* [Function]

The function `current-kill` rotates the yanking pointer, which designates the "front" of the kill ring, by *n* places (from newer kills to older ones), and returns the text at that place in the ring.

If the optional second argument *do-not-move* is non-`nil`, then `current-kill` doesn't alter the yanking pointer; it just returns the *n*th kill, counting from the current yanking pointer.

If *n* is zero, indicating a request for the latest kill, `current-kill` calls the value of `interprogram-paste-function` (documented below) before consulting the kill ring. If that value is a function and calling it returns a string or a list of several string, `current-kill` pushes the strings onto the kill ring and returns the first string. It also sets the yanking pointer to point to the kill-ring entry of the first string returned by `interprogram-paste-function`, regardless of the value of *do-not-move*. Otherwise,

`current-kill` does not treat a zero value for *n* specially: it returns the entry pointed at by the yanking pointer and does not move the yanking pointer.

`kill-new` *string* **&optional** *replace* [Function]
This function pushes the text *string* onto the kill ring and makes the yanking pointer point to it. It discards the oldest entry if appropriate. It also invokes the value of `interprogram-cut-function` (see below).

If *replace* is non-`nil`, then `kill-new` replaces the first element of the kill ring with *string*, rather than pushing *string* onto the kill ring.

`kill-append` *string before-p* [Function]
This function appends the text *string* to the first entry in the kill ring and makes the yanking pointer point to the combined entry. Normally *string* goes at the end of the entry, but if *before-p* is non-`nil`, it goes at the beginning. This function also invokes the value of `interprogram-cut-function` (see below).

`interprogram-paste-function` [Variable]
This variable provides a way of transferring killed text from other programs, when you are using a window system. Its value should be `nil` or a function of no arguments.

If the value is a function, `current-kill` calls it to get the "most recent kill". If the function returns a non-`nil` value, then that value is used as the "most recent kill". If it returns `nil`, then the front of the kill ring is used.

To facilitate support for window systems that support multiple selections, this function may also return a list of strings. In that case, the first string is used as the "most recent kill", and all the other strings are pushed onto the kill ring, for easy access by `yank-pop`.

The normal use of this function is to get the window system's clipboard as the most recent kill, even if the selection belongs to another application. See Section 18.18 [Window System Selections], page 366. However, if the clipboard contents come from the current Emacs session, this function should return `nil`.

`interprogram-cut-function` [Variable]
This variable provides a way of communicating killed text to other programs, when you are using a window system. Its value should be `nil` or a function of one required argument.

If the value is a function, `kill-new` and `kill-append` call it with the new first element of the kill ring as the argument.

The normal use of this function is to put newly killed text in the window system's clipboard. See Section 18.18 [Window System Selections], page 366.

### 22.8.6 Internals of the Kill Ring

The variable `kill-ring` holds the kill ring contents, in the form of a list of strings. The most recent kill is always at the front of the list.

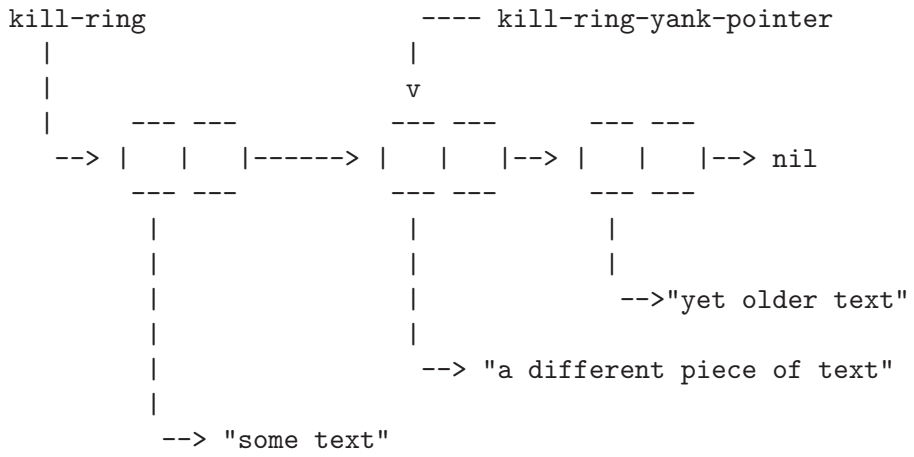
The `kill-ring-yank-pointer` variable points to a link in the kill ring list, whose CAR is the text to yank next. We say it identifies the "front" of the ring. Moving `kill-ring-yank-pointer` to a different link is called *rotating the kill ring*. We call the kill ring a

"ring" because the functions that move the yank pointer wrap around from the end of the list to the beginning, or vice-versa. Rotation of the kill ring is virtual; it does not change the value of `kill-ring`.

Both `kill-ring` and `kill-ring-yank-pointer` are Lisp variables whose values are normally lists. The word "pointer" in the name of the `kill-ring-yank-pointer` indicates that the variable's purpose is to identify one element of the list for use by the next yank command.

The value of `kill-ring-yank-pointer` is always `eq` to one of the links in the kill ring list. The element it identifies is the CAR of that link. Kill commands, which change the kill ring, also set this variable to the value of `kill-ring`. The effect is to rotate the ring so that the newly killed text is at the front.

Here is a diagram that shows the variable `kill-ring-yank-pointer` pointing to the second entry in the kill ring `("some text" "a different piece of text" "yet older text")`.




This state of affairs might occur after `C-y` (`yank`) immediately followed by `M-y` (`yank-pop`).

`kill-ring` [Variable]
: This variable holds the list of killed text sequences, most recently killed first.

`kill-ring-yank-pointer` [Variable]
: This variable's value indicates which element of the kill ring is at the "front" of the ring for yanking. More precisely, the value is a tail of the value of `kill-ring`, and its CAR is the kill string that `C-y` should yank.

`kill-ring-max` [User Option]
: The value of this variable is the maximum length to which the kill ring can grow, before elements are thrown away at the end. The default value for `kill-ring-max` is 60.

## 22.9 Undo

Most buffers have an *undo list*, which records all changes made to the buffer's text so that they can be undone. (The buffers that don't have one are usually special-purpose buffers for which Emacs assumes that undoing is not useful. In particular, any buffer whose name begins with a space has its undo recording off by default; see Section 16.3 [Buffer Names],

page 275.) All the primitives that modify the text in the buffer automatically add elements to the front of the undo list, which is in the variable `buffer-undo-list`.

`buffer-undo-list` [Variable]
: This buffer-local variable's value is the undo list of the current buffer. A value of `t` disables the recording of undo information.

Here are the kinds of elements an undo list can have:

*position*
: This kind of element records a previous value of point; undoing this element moves point to *position*. Ordinary cursor motion does not make any sort of undo record, but deletion operations use these entries to record where point was before the command.

(*beg* . *end*)
: This kind of element indicates how to delete text that was inserted. Upon insertion, the text occupied the range *beg*–*end* in the buffer.

(*text* . *position*)
: This kind of element indicates how to reinsert text that was deleted. The deleted text itself is the string *text*. The place to reinsert it is (`abs` *position*). If *position* is positive, point was at the beginning of the deleted text, otherwise it was at the end.

(`t` *sec-high sec-low microsec picosec*)
: This kind of element indicates that an unmodified buffer became modified. The list (*sec-high sec-low microsec picosec*) represents the visited file's modification time as of when it was previously visited or saved, using the same format as `current-time`; see ⟨undefined⟩ [Time of Day], page ⟨undefined⟩. `primitive-undo` uses those values to determine whether to mark the buffer as unmodified once again; it does so only if the file's modification time matches those numbers.

(`nil` *property value beg* . *end*)
: This kind of element records a change in a text property. Here's how you might undo the change:

```
(put-text-property beg end property value)
```

(*marker* . *adjustment*)
: This kind of element records the fact that the marker *marker* was relocated due to deletion of surrounding text, and that it moved *adjustment* character positions. Undoing this element moves *marker* − *adjustment* characters.

(`apply` *funname* . *args*)
: This is an extensible undo item, which is undone by calling *funname* with arguments *args*.

(`apply` *delta beg end funname* . *args*)
: This is an extensible undo item, which records a change limited to the range *beg* to *end*, which increased the size of the buffer by *delta*. It is undone by calling *funname* with arguments *args*.

  This kind of element enables undo limited to a region to determine whether the element pertains to that region.

`nil` This element is a boundary. The elements between two boundaries are called a *change group*; normally, each change group corresponds to one keyboard command, and undo commands normally undo an entire group as a unit.

`undo-boundary` [Function]
This function places a boundary element in the undo list. The undo command stops at such a boundary, and successive undo commands undo to earlier and earlier boundaries. This function returns `nil`.

The editor command loop automatically calls `undo-boundary` just before executing each key sequence, so that each undo normally undoes the effects of one command. As an exception, the command `self-insert-command`, which produces self-inserting input characters (see Section 22.5 [Commands for Insertion], page 459), may remove the boundary inserted by the command loop: a boundary is accepted for the first such character, the next 19 consecutive self-inserting input characters do not have boundaries, and then the 20th does; and so on as long as the self-inserting characters continue. Hence, sequences of consecutive character insertions can be undone as a group.

All buffer modifications add a boundary whenever the previous undoable change was made in some other buffer. This is to ensure that each command makes a boundary in each buffer where it makes changes.

Calling this function explicitly is useful for splitting the effects of a command into more than one unit. For example, `query-replace` calls `undo-boundary` after each replacement, so that the user can undo individual replacements one by one.

`undo-in-progress` [Variable]
This variable is normally `nil`, but the undo commands bind it to `t`. This is so that various kinds of change hooks can tell when they're being called for the sake of undoing.

`primitive-undo` *count list* [Function]
This is the basic function for undoing elements of an undo list. It undoes the first *count* elements of *list*, returning the rest of *list*.

`primitive-undo` adds elements to the buffer's undo list when it changes the buffer. Undo commands avoid confusion by saving the undo list value at the beginning of a sequence of undo operations. Then the undo operations use and update the saved value. The new elements added by undoing are not part of this saved value, so they don't interfere with continuing to undo.

This function does not bind `undo-in-progress`.

## 22.10 Maintaining Undo Lists

This section describes how to enable and disable undo information for a given buffer. It also explains how the undo list is truncated automatically so it doesn't get too big.

Recording of undo information in a newly created buffer is normally enabled to start with; but if the buffer name starts with a space, the undo recording is initially disabled. You can explicitly enable or disable undo recording with the following two functions, or by setting `buffer-undo-list` yourself.

`buffer-enable-undo` **&optional** *buffer-or-name* [Command]
: This command enables recording undo information for buffer *buffer-or-name*, so that subsequent changes can be undone. If no argument is supplied, then the current buffer is used. This function does nothing if undo recording is already enabled in the buffer. It returns `nil`.

  In an interactive call, *buffer-or-name* is the current buffer. You cannot specify any other buffer.

`buffer-disable-undo` **&optional** *buffer-or-name* [Command]
: This function discards the undo list of *buffer-or-name*, and disables further recording of undo information. As a result, it is no longer possible to undo either previous changes or any subsequent changes. If the undo list of *buffer-or-name* is already disabled, this function has no effect.

  This function returns `nil`.

As editing continues, undo lists get longer and longer. To prevent them from using up all available memory space, garbage collection trims them back to size limits you can set. (For this purpose, the "size" of an undo list measures the cons cells that make up the list, plus the strings of deleted text.) Three variables control the range of acceptable sizes: `undo-limit`, `undo-strong-limit` and `undo-outer-limit`. In these variables, size is counted as the number of bytes occupied, which includes both saved text and other data.

`undo-limit` [User Option]
: This is the soft limit for the acceptable size of an undo list. The change group at which this size is exceeded is the last one kept.

`undo-strong-limit` [User Option]
: This is the upper limit for the acceptable size of an undo list. The change group at which this size is exceeded is discarded itself (along with all older change groups). There is one exception: the very latest change group is only discarded if it exceeds `undo-outer-limit`.

`undo-outer-limit` [User Option]
: If at garbage collection time the undo info for the current command exceeds this limit, Emacs discards the info and displays a warning. This is a last ditch limit to prevent memory overflow.

`undo-ask-before-discard` [User Option]
: If this variable is non-`nil`, when the undo info exceeds `undo-outer-limit`, Emacs asks in the echo area whether to discard the info. The default value is `nil`, which means to discard it automatically.

  This option is mainly intended for debugging. Garbage collection is inhibited while the question is asked, which means that Emacs might leak memory if the user waits too long before answering the question.

## 22.11 Filling

*Filling* means adjusting the lengths of lines (by moving the line breaks) so that they are nearly (but no greater than) a specified maximum width. Additionally, lines can be *justified*, which means inserting spaces to make the left and/or right margins line up precisely. The width is controlled by the variable `fill-column`. For ease of reading, lines should be no longer than 70 or so columns.

You can use Auto Fill mode (see Section 22.14 [Auto Filling], page 478) to fill text automatically as you insert it, but changes to existing text may leave it improperly filled. Then you must fill the text explicitly.

Most of the commands in this section return values that are not meaningful. All the functions that do filling take note of the current left margin, current right margin, and current justification style (see Section 22.12 [Margins], page 475). If the current justification style is `none`, the filling functions don't actually do anything.

Several of the filling functions have an argument *justify*. If it is non-`nil`, that requests some kind of justification. It can be `left`, `right`, `full`, or `center`, to request a specific style of justification. If it is `t`, that means to use the current justification style for this part of the text (see `current-justification`, below). Any other value is treated as `full`.

When you call the filling functions interactively, using a prefix argument implies the value `full` for *justify*.

`fill-paragraph` **&optional** *justify region* [Command]
: This command fills the paragraph at or after point. If *justify* is non-`nil`, each line is justified as well. It uses the ordinary paragraph motion commands to find paragraph boundaries. See Section "Paragraphs" in *The GNU Emacs Manual*.

  When *region* is non-`nil`, then if Transient Mark mode is enabled and the mark is active, this command calls `fill-region` to fill all the paragraphs in the region, instead of filling only the current paragraph. When this command is called interactively, *region* is `t`.

`fill-region` *start end* **&optional** *justify nosqueeze to-eop* [Command]
: This command fills each of the paragraphs in the region from *start* to *end*. It justifies as well if *justify* is non-`nil`.

  If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *to-eop* is non-`nil`, that means to keep filling to the end of the paragraph—or the next hard newline, if `use-hard-newlines` is enabled (see below).

  The variable `paragraph-separate` controls how to distinguish paragraphs. See ⟨undefined⟩ [Standard Regexps], page ⟨undefined⟩.

`fill-individual-paragraphs` *start end* **&optional** *justify citation-regexp* [Command]
: This command fills each paragraph in the region according to its individual fill prefix. Thus, if the lines of a paragraph were indented with spaces, the filled paragraph will remain indented in the same fashion.

  The first two arguments, *start* and *end*, are the beginning and end of the region to be filled. The third and fourth arguments, *justify* and *citation-regexp*, are optional. If *justify* is non-`nil`, the paragraphs are justified as well as filled. If *citation-regexp* is

non-`nil`, it means the function is operating on a mail message and therefore should not fill the header lines. If *citation-regexp* is a string, it is used as a regular expression; if it matches the beginning of a line, that line is treated as a citation marker.

Ordinarily, `fill-individual-paragraphs` regards each change in indentation as starting a new paragraph. If `fill-individual-varying-indent` is non-`nil`, then only separator lines separate paragraphs. That mode can handle indented paragraphs with additional indentation on the first line.

`fill-individual-varying-indent` [User Option]
This variable alters the action of `fill-individual-paragraphs` as described above.

`fill-region-as-paragraph` *start end* **&optional** *justify nosqueeze squeeze-after* [Command]
This command considers a region of text as a single paragraph and fills it. If the region was made up of many paragraphs, the blank lines between paragraphs are removed. This function justifies as well as filling when *justify* is non-`nil`.

If *nosqueeze* is non-`nil`, that means to leave whitespace other than line breaks untouched. If *squeeze-after* is non-`nil`, it specifies a position in the region, and means don't canonicalize spaces before that position.

In Adaptive Fill mode, this command calls `fill-context-prefix` to choose a fill prefix by default. See Section 22.13 [Adaptive Fill], page 477.

`justify-current-line` **&optional** *how eop nosqueeze* [Command]
This command inserts spaces between the words of the current line so that the line ends exactly at `fill-column`. It returns `nil`.

The argument *how*, if non-`nil` specifies explicitly the style of justification. It can be `left`, `right`, `full`, `center`, or `none`. If it is `t`, that means to do follow specified justification style (see `current-justification`, below). `nil` means to do full justification.

If *eop* is non-`nil`, that means do only left-justification if `current-justification` specifies full justification. This is used for the last line of a paragraph; even if the paragraph as a whole is fully justified, the last line should not be.

If *nosqueeze* is non-`nil`, that means do not change interior whitespace.

`default-justification` [User Option]
This variable's value specifies the style of justification to use for text that doesn't specify a style with a text property. The possible values are `left`, `right`, `full`, `center`, or `none`. The default value is `left`.

`current-justification` [Function]
This function returns the proper justification style to use for filling the text around point.

This returns the value of the `justification` text property at point, or the variable *default-justification* if there is no such text property. However, it returns `nil` rather than `none` to mean "don't justify".

`sentence-end-double-space` [User Option]
If this variable is non-`nil`, a period followed by just one space does not count as the end of a sentence, and the filling functions avoid breaking the line at such a place.

`sentence-end-without-period` [User Option]
If this variable is non-`nil`, a sentence can end without a period. This is used for languages like Thai, where sentences end with a double space but without a period.

`sentence-end-without-space` [User Option]
If this variable is non-`nil`, it should be a string of characters that can end a sentence without following spaces.

`fill-paragraph-function` [Variable]
This variable provides a way to override the filling of paragraphs. If its value is non-`nil`, `fill-paragraph` calls this function to do the work. If the function returns a non-`nil` value, `fill-paragraph` assumes the job is done, and immediately returns that value.

The usual use of this feature is to fill comments in programming language modes. If the function needs to fill a paragraph in the usual way, it can do so as follows:

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

`fill-forward-paragraph-function` [Variable]
This variable provides a way to override how the filling functions, such as `fill-region` and `fill-paragraph`, move forward to the next paragraph. Its value should be a function, which is called with a single argument *n*, the number of paragraphs to move, and should return the difference between *n* and the number of paragraphs actually moved. The default value of this variable is `forward-paragraph`. See Section "Paragraphs" in *The GNU Emacs Manual*.

`use-hard-newlines` [Variable]
If this variable is non-`nil`, the filling functions do not delete newlines that have the `hard` text property. These "hard newlines" act as paragraph separators.

## 22.12 Margins for Filling

`fill-prefix` [User Option]
This buffer-local variable, if non-`nil`, specifies a string of text that appears at the beginning of normal text lines and should be disregarded when filling them. Any line that fails to start with the fill prefix is considered the start of a paragraph; so is any line that starts with the fill prefix followed by additional whitespace. Lines that start with the fill prefix but no additional whitespace are ordinary text lines that can be filled together. The resulting filled lines also start with the fill prefix.

The fill prefix follows the left margin whitespace, if any.

`fill-column` [User Option]
This buffer-local variable specifies the maximum width of filled lines. Its value should be an integer, which is a number of columns. All the filling, justification, and centering

commands are affected by this variable, including Auto Fill mode (see Section 22.14 [Auto Filling], page 478).

As a practical matter, if you are writing text for other people to read, you should set `fill-column` to no more than 70. Otherwise the line will be too long for people to read comfortably, and this can make the text seem clumsy.

The default value for `fill-column` is 70.

`set-left-margin` *from to margin* [Command]
This sets the `left-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

`set-right-margin` *from to margin* [Command]
This sets the `right-margin` property on the text from *from* to *to* to the value *margin*. If Auto Fill mode is enabled, this command also refills the region to fit the new margin.

`current-left-margin` [Function]
This function returns the proper left margin value to use for filling the text around point. The value is the sum of the `left-margin` property of the character at the start of the current line (or zero if none), and the value of the variable `left-margin`.

`current-fill-column` [Function]
This function returns the proper fill column value to use for filling the text around point. The value is the value of the `fill-column` variable, minus the value of the `right-margin` property of the character after point.

`move-to-left-margin` **&optional** *n force* [Command]
This function moves point to the left margin of the current line. The column moved to is determined by calling the function `current-left-margin`. If the argument *n* is non-`nil`, `move-to-left-margin` moves forward $n-1$ lines first.

If *force* is non-`nil`, that says to fix the line's indentation if that doesn't match the left margin value.

`delete-to-left-margin` **&optional** *from to* [Function]
This function removes left margin indentation from the text between *from* and *to*. The amount of indentation to delete is determined by calling `current-left-margin`. In no case does this function delete non-whitespace. If *from* and *to* are omitted, they default to the whole buffer.

`indent-to-left-margin` [Function]
This function adjusts the indentation at the beginning of the current line to the value specified by the variable `left-margin`. (That may involve either inserting or deleting whitespace.) This function is value of `indent-line-function` in Paragraph-Indent Text mode.

`left-margin` [User Option]
This variable specifies the base left margin column. In Fundamental mode, `C-j` indents to this column. This variable automatically becomes buffer-local when set in any fashion.

`fill-nobreak-predicate` [User Option]
: This variable gives major modes a way to specify not to break a line at certain places. Its value should be a list of functions. Whenever filling considers breaking the line at a certain place in the buffer, it calls each of these functions with no arguments and with point located at that place. If any of the functions returns non-`nil`, then the line won't be broken there.

## 22.13 Adaptive Fill Mode

When *Adaptive Fill Mode* is enabled, Emacs determines the fill prefix automatically from the text in each paragraph being filled rather than using a predetermined value. During filling, this fill prefix gets inserted at the start of the second and subsequent lines of the paragraph as described in Section 22.11 [Filling], page 473, and in Section 22.14 [Auto Filling], page 478.

`adaptive-fill-mode` [User Option]
: Adaptive Fill mode is enabled when this variable is non-`nil`. It is `t` by default.

`fill-context-prefix` *from to* [Function]
: This function implements the heart of Adaptive Fill mode; it chooses a fill prefix based on the text between *from* and *to*, typically the start and end of a paragraph. It does this by looking at the first two lines of the paragraph, based on the variables described below.

  Usually, this function returns the fill prefix, a string. However, before doing this, the function makes a final check (not specially mentioned in the following) that a line starting with this prefix wouldn't look like the start of a paragraph. Should this happen, the function signals the anomaly by returning `nil` instead.

  In detail, `fill-context-prefix` does this:

  1. It takes a candidate for the fill prefix from the first line—it tries first the function in `adaptive-fill-function` (if any), then the regular expression `adaptive-fill-regexp` (see below). The first non-`nil` result of these, or the empty string if they're both `nil`, becomes the first line's candidate.
  2. If the paragraph has as yet only one line, the function tests the validity of the prefix candidate just found. The function then returns the candidate if it's valid, or a string of spaces otherwise. (see the description of `adaptive-fill-first-line-regexp` below).
  3. When the paragraph already has two lines, the function next looks for a prefix candidate on the second line, in just the same way it did for the first line. If it doesn't find one, it returns `nil`.
  4. The function now compares the two candidate prefixes heuristically: if the non-whitespace characters in the line 2 candidate occur in the same order in the line 1 candidate, the function returns the line 2 candidate. Otherwise, it returns the largest initial substring which is common to both candidates (which might be the empty string).

`adaptive-fill-regexp` [User Option]
Adaptive Fill mode matches this regular expression against the text starting after the left margin whitespace (if any) on a line; the characters it matches are that line's candidate for the fill prefix.

The default value matches whitespace with certain punctuation characters intermingled.

`adaptive-fill-first-line-regexp` [User Option]
Used only in one-line paragraphs, this regular expression acts as an additional check of the validity of the one available candidate fill prefix: the candidate must match this regular expression, or match `comment-start-skip`. If it doesn't, `fill-context-prefix` replaces the candidate with a string of spaces "of the same width" as it.

The default value of this variable is `"\\‘[ \t]*\\’"`, which matches only a string of whitespace. The effect of this default is to force the fill prefixes found in one-line paragraphs always to be pure whitespace.

`adaptive-fill-function` [User Option]
You can specify more complex ways of choosing a fill prefix automatically by setting this variable to a function. The function is called with point after the left margin (if any) of a line, and it must preserve point. It should return either "that line's" fill prefix or `nil`, meaning it has failed to determine a prefix.

## 22.14 Auto Filling

Auto Fill mode is a minor mode that fills lines automatically as text is inserted. This section describes the hook used by Auto Fill mode. For a description of functions that you can call explicitly to fill and justify existing text, see Section 22.11 [Filling], page 473.

Auto Fill mode also enables the functions that change the margins and justification style to refill portions of the text. See Section 22.12 [Margins], page 475.

`auto-fill-function` [Variable]
The value of this buffer-local variable should be a function (of no arguments) to be called after self-inserting a character from the table `auto-fill-chars`. It may be `nil`, in which case nothing special is done in that case.

The value of `auto-fill-function` is `do-auto-fill` when Auto-Fill mode is enabled. That is a function whose sole purpose is to implement the usual strategy for breaking a line.

`normal-auto-fill-function` [Variable]
This variable specifies the function to use for `auto-fill-function`, if and when Auto Fill is turned on. Major modes can set buffer-local values for this variable to alter how Auto Fill works.

`auto-fill-chars` [Variable]
A char table of characters which invoke `auto-fill-function` when self-inserted—space and newline in most language environments. They have an entry `t` in the table.

## 22.15 Sorting Text

The sorting functions described in this section all rearrange text in a buffer. This is in contrast to the function `sort`, which rearranges the order of the elements of a list (see ⟨undefined⟩ [Rearrangement], page ⟨undefined⟩). The values returned by these functions are not meaningful.

`sort-subr` *reverse nextrecfun endrecfun* **&optional** *startkeyfun endkeyfun predicate* [Function]

This function is the general text-sorting routine that subdivides a buffer into records and then sorts them. Most of the commands in this section use this function.

To understand how `sort-subr` works, consider the whole accessible portion of the buffer as being divided into disjoint pieces called *sort records*. The records may or may not be contiguous, but they must not overlap. A portion of each sort record (perhaps all of it) is designated as the sort key. Sorting rearranges the records in order by their sort keys.

Usually, the records are rearranged in order of ascending sort key. If the first argument to the `sort-subr` function, *reverse*, is non-`nil`, the sort records are rearranged in order of descending sort key.

The next four arguments to `sort-subr` are functions that are called to move point across a sort record. They are called many times from within `sort-subr`.

1. *nextrecfun* is called with point at the end of a record. This function moves point to the start of the next record. The first record is assumed to start at the position of point when `sort-subr` is called. Therefore, you should usually move point to the beginning of the buffer before calling `sort-subr`.

   This function can indicate there are no more sort records by leaving point at the end of the buffer.

2. *endrecfun* is called with point within a record. It moves point to the end of the record.

3. *startkeyfun* is called to move point from the start of a record to the start of the sort key. This argument is optional; if it is omitted, the whole record is the sort key. If supplied, the function should either return a non-`nil` value to be used as the sort key, or return `nil` to indicate that the sort key is in the buffer starting at point. In the latter case, *endkeyfun* is called to find the end of the sort key.

4. *endkeyfun* is called to move point from the start of the sort key to the end of the sort key. This argument is optional. If *startkeyfun* returns `nil` and this argument is omitted (or `nil`), then the sort key extends to the end of the record. There is no need for *endkeyfun* if *startkeyfun* returns a non-`nil` value.

The argument *predicate* is the function to use to compare keys. If keys are numbers, it defaults to `<`; otherwise it defaults to `string<`.

As an example of `sort-subr`, here is the complete function definition for `sort-lines`:

```
;; Note that the first two lines of doc string
;; are effectively one line when viewed by a user.
(defun sort-lines (reverse beg end)
  "Sort lines in region alphabetically;\
 argument means descending order.
Called from a program, there are three arguments:
REVERSE (non-nil means reverse order),\
 BEG and END (region to sort).
The variable `sort-fold-case' determines\
 whether alphabetic case affects
the sort order."
  (interactive "P\nr")
  (save-excursion
    (save-restriction
      (narrow-to-region beg end)
      (goto-char (point-min))
      (let ((inhibit-field-text-motion t))
        (sort-subr reverse 'forward-line 'end-of-line)))))
```

Here `forward-line` moves point to the start of the next record, and `end-of-line` moves point to the end of record. We do not pass the arguments *startkeyfun* and *endkeyfun*, because the entire record is used as the sort key.

The `sort-paragraphs` function is very much the same, except that its `sort-subr` call looks like this:

```
(sort-subr reverse
           (function
             (lambda ()
               (while (and (not (eobp))
                           (looking-at paragraph-separate))
                 (forward-line 1))))
           'forward-paragraph)
```

Markers pointing into any sort records are left with no useful position after `sort-subr` returns.

`sort-fold-case` [User Option]

If this variable is non-`nil`, `sort-subr` and the other buffer sorting functions ignore case when comparing strings.

`sort-regexp-fields` *reverse record-regexp key-regexp start end* [Command]

This command sorts the region between *start* and *end* alphabetically as specified by *record-regexp* and *key-regexp*. If *reverse* is a negative integer, then sorting is in reverse order.

Alphabetical sorting means that two sort keys are compared by comparing the first characters of each, the second characters of each, and so on. If a mismatch is found, it means that the sort keys are unequal; the sort key whose character is less at the point of first mismatch is the lesser sort key. The individual characters are compared according to their numerical character codes in the Emacs character set.

The value of the *record-regexp* argument specifies how to divide the buffer into sort records. At the end of each record, a search is done for this regular expression, and the text that matches it is taken as the next record. For example, the regular expression '`^.+$`', which matches lines with at least one character besides a newline, would make each such line into a sort record. See ⟨undefined⟩ [Regular Expressions], page ⟨undefined⟩, for a description of the syntax and meaning of regular expressions.

The value of the *key-regexp* argument specifies what part of each record is the sort key. The *key-regexp* could match the whole record, or only a part. In the latter case, the rest of the record has no effect on the sorted order of records, but it is carried along when the record moves to its new position.

The *key-regexp* argument can refer to the text matched by a subexpression of *record-regexp*, or it can be a regular expression on its own.

If *key-regexp* is:

'`\`*digit*' then the text matched by the *digit*th '`\(...\)`' parenthesis grouping in *record-regexp* is the sort key.

'`\&`' then the whole record is the sort key.

a regular expression
: then `sort-regexp-fields` searches for a match for the regular expression within the record. If such a match is found, it is the sort key. If there is no match for *key-regexp* within a record then that record is ignored, which means its position in the buffer is not changed. (The other records may move around it.)

For example, if you plan to sort all the lines in the region by the first word on each line starting with the letter '`f`', you should set *record-regexp* to '`^.*$`' and set *key-regexp* to '`\<f\w*\>`'. The resulting expression looks like this:

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

If you call `sort-regexp-fields` interactively, it prompts for *record-regexp* and *key-regexp* in the minibuffer.

`sort-lines` *reverse start end* [Command]
: This command alphabetically sorts lines in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-paragraphs` *reverse start end* [Command]
: This command alphabetically sorts paragraphs in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-pages` *reverse start end* [Command]
: This command alphabetically sorts pages in the region between *start* and *end*. If *reverse* is non-`nil`, the sort is in reverse order.

`sort-fields` *field start end* [Command]
: This command sorts lines in the region between *start* and *end*, comparing them alphabetically by the *field*th field of each line. Fields are separated by whitespace

and numbered starting from 1. If *field* is negative, sorting is by the −*field*th field from the end of the line. This command is useful for sorting tables.

`sort-numeric-fields` *field start end* [Command]
This command sorts lines in the region between *start* and *end*, comparing them numerically by the *field*th field of each line. Fields are separated by whitespace and numbered starting from 1. The specified field must contain a number in each line of the region. Numbers starting with 0 are treated as octal, and numbers starting with '`0x`' are treated as hexadecimal.

If *field* is negative, sorting is by the −*field*th field from the end of the line. This command is useful for sorting tables.

`sort-numeric-base` [User Option]
This variable specifies the default radix for `sort-numeric-fields` to parse numbers.

`sort-columns` *reverse* **&optional** *beg end* [Command]
This command sorts the lines in the region between *beg* and *end*, comparing them alphabetically by a certain range of columns. The column positions of *beg* and *end* bound the range of columns to sort on.

If *reverse* is non-`nil`, the sort is in reverse order.

One unusual thing about this command is that the entire line containing position *beg*, and the entire line containing position *end*, are included in the region sorted.

Note that `sort-columns` rejects text that contains tabs, because tabs could be split across the specified columns. Use `M-x untabify` to convert tabs to spaces before sorting.

When possible, this command actually works by calling the `sort` utility program.

## 22.16 Counting Columns

The column functions convert between a character position (counting characters from the beginning of the buffer) and a column position (counting screen characters from the beginning of a line).

These functions count each character according to the number of columns it occupies on the screen. This means control characters count as occupying 2 or 4 columns, depending upon the value of `ctl-arrow`, and tabs count as occupying a number of columns that depends on the value of `tab-width` and on the column where the tab begins. See Section 11.20.1 [Usual Display], page 189.

Column number computations ignore the width of the window and the amount of horizontal scrolling. Consequently, a column value can be arbitrarily high. The first (or leftmost) column is numbered 0. They also ignore overlays and text properties, aside from invisibility.

`current-column` [Function]
This function returns the horizontal position of point, measured in columns, counting from 0 at the left margin. The column position is the sum of the widths of all the displayed representations of the characters between the start of the current line and point.

For an example of using `current-column`, see the description of `count-lines` in ⟨undefined⟩ [Text Lines], page ⟨undefined⟩.

`move-to-column` *column* **&optional** *force* [Command]

This function moves point to *column* in the current line. The calculation of *column* takes into account the widths of the displayed representations of the characters between the start of the line and point.

When called interactively, *column* is the value of prefix numeric argument. If *column* is not an integer, an error is signaled.

If column *column* is beyond the end of the line, point moves to the end of the line. If *column* is negative, point moves to the beginning of the line.

If it is impossible to move to column *column* because that is in the middle of a multi-column character such as a tab, point moves to the end of that character. However, if *force* is non-`nil`, and *column* is in the middle of a tab, then `move-to-column` converts the tab into spaces so that it can move precisely to column *column*. Other multi-column characters can cause anomalies despite *force*, since there is no way to split them.

The argument *force* also has an effect if the line isn't long enough to reach column *column*; if it is `t`, that means to add whitespace at the end of the line to reach that column.

The return value is the column number actually moved to.

## 22.17 Indentation

The indentation functions are used to examine, move to, and change whitespace that is at the beginning of a line. Some of the functions can also change whitespace elsewhere on a line. Columns and indentation count from zero at the left margin.

### 22.17.1 Indentation Primitives

This section describes the primitive functions used to count and insert indentation. The functions in the following sections use these primitives. See Section 11.10 [Width], page 135, for related functions.

`current-indentation` [Function]

This function returns the indentation of the current line, which is the horizontal position of the first nonblank character. If the contents are entirely blank, then this is the horizontal position of the end of the line.

`indent-to` *column* **&optional** *minimum* [Command]

This function indents from point with tabs and spaces until *column* is reached. If *minimum* is specified and non-`nil`, then at least that many spaces are inserted even if this requires going beyond *column*. Otherwise the function does nothing if point is already beyond *column*. The value is the column at which the inserted indentation ends.

The inserted whitespace characters inherit text properties from the surrounding text (usually, from the preceding text only). See Section 22.19.6 [Sticky Properties], page 500.

`indent-tabs-mode` [User Option]
: If this variable is non-`nil`, indentation functions can insert tabs as well as spaces. Otherwise, they insert only spaces. Setting this variable automatically makes it buffer-local in the current buffer.

### 22.17.2 Indentation Controlled by Major Mode

An important function of each major mode is to customize the TAB key to indent properly for the language being edited. This section describes the mechanism of the TAB key and how to control it. The functions in this section return unpredictable values.

`indent-for-tab-command` **&optional** *rigid* [Command]
: This is the command bound to TAB in most editing modes. Its usual action is to indent the current line, but it can alternatively insert a tab character or indent a region.

  Here is what it does:

  - First, it checks whether Transient Mark mode is enabled and the region is active. If so, it called `indent-region` to indent all the text in the region (see Section 22.17.3 [Region Indent], page 485).
  - Otherwise, if the indentation function in `indent-line-function` is `indent-to-left-margin` (a trivial command that inserts a tab character), or if the variable `tab-always-indent` specifies that a tab character ought to be inserted (see below), then it inserts a tab character.
  - Otherwise, it indents the current line; this is done by calling the function in `indent-line-function`. If the line is already indented, and the value of `tab-always-indent` is `complete` (see below), it tries completing the text at point.

  If *rigid* is non-`nil` (interactively, with a prefix argument), then after this command indents a line or inserts a tab, it also rigidly indents the entire balanced expression which starts at the beginning of the current line, in order to reflect the new indentation. This argument is ignored if the command indents the region.

`indent-line-function` [Variable]
: This variable's value is the function to be used by `indent-for-tab-command`, and various other indentation commands, to indent the current line. It is usually assigned by the major mode; for instance, Lisp mode sets it to `lisp-indent-line`, C mode sets it to `c-indent-line`, and so on. The default value is `indent-relative`. See Section 20.7 [Auto-Indentation], page 440.

`indent-according-to-mode` [Command]
: This command calls the function in `indent-line-function` to indent the current line in a way appropriate for the current major mode.

`newline-and-indent` [Command]
: This function inserts a newline, then indents the new line (the one following the newline just inserted) according to the major mode. It does indentation by calling `indent-according-to-mode`.

`reindent-then-newline-and-indent` [Command]
: This command reindents the current line, inserts a newline at point, and then indents the new line (the one following the newline just inserted). It does indentation on both lines by calling `indent-according-to-mode`.

`tab-always-indent` [User Option]
: This variable can be used to customize the behavior of the TAB (`indent-for-tab-command`) command. If the value is `t` (the default), the command normally just indents the current line. If the value is `nil`, the command indents the current line only if point is at the left margin or in the line's indentation; otherwise, it inserts a tab character. If the value is `complete`, the command first tries to indent the current line, and if the line was already indented, it calls `completion-at-point` to complete the text at point (see ⟨undefined⟩ [Completion in Buffers], page ⟨undefined⟩).

### 22.17.3 Indenting an Entire Region

This section describes commands that indent all the lines in the region. They return unpredictable values.

`indent-region` *start end* **&optional** *to-column* [Command]
: This command indents each nonblank line starting between *start* (inclusive) and *end* (exclusive). If *to-column* is `nil`, `indent-region` indents each nonblank line by calling the current mode's indentation function, the value of `indent-line-function`.

  If *to-column* is non-`nil`, it should be an integer specifying the number of columns of indentation; then this function gives each line exactly that much indentation, by either adding or deleting whitespace.

  If there is a fill prefix, `indent-region` indents each line by making it start with the fill prefix.

`indent-region-function` [Variable]
: The value of this variable is a function that can be used by `indent-region` as a short cut. It should take two arguments, the start and end of the region. You should design the function so that it will produce the same results as indenting the lines of the region one by one, but presumably faster.

  If the value is `nil`, there is no short cut, and `indent-region` actually works line by line.

  A short-cut function is useful in modes such as C mode and Lisp mode, where the `indent-line-function` must scan from the beginning of the function definition: applying it to each line would be quadratic in time. The short cut can update the scan information as it moves through the lines indenting them; this takes linear time. In a mode where indenting a line individually is fast, there is no need for a short cut.

  `indent-region` with a non-`nil` argument *to-column* has a different meaning and does not use this variable.

`indent-rigidly` *start end count* [Command]
: This command indents all lines starting between *start* (inclusive) and *end* (exclusive) sideways by *count* columns. This "preserves the shape" of the affected region, moving

it as a rigid unit. Consequently, this command is useful not only for indenting regions of unindented text, but also for indenting regions of formatted code.

For example, if *count* is 3, this command adds 3 columns of indentation to each of the lines beginning in the region specified.

In Mail mode, `C-c C-y` (`mail-yank-original`) uses `indent-rigidly` to indent the text copied from the message being replied to.

`indent-code-rigidly` *start end columns* **&optional** *nochange-regexp* [Command]
This is like `indent-rigidly`, except that it doesn't alter lines that start within strings or comments.

In addition, it doesn't alter a line if *nochange-regexp* matches at the beginning of the line (if *nochange-regexp* is non-`nil`).

### 22.17.4 Indentation Relative to Previous Lines

This section describes two commands that indent the current line based on the contents of previous lines.

`indent-relative` **&optional** *unindented-ok* [Command]
This command inserts whitespace at point, extending to the same column as the next *indent point* of the previous nonblank line. An indent point is a non-whitespace character following whitespace. The next indent point is the first one at a column greater than the current column of point. For example, if point is underneath and to the left of the first non-blank character of a line of text, it moves to that column by inserting whitespace.

If the previous nonblank line has no next indent point (i.e., none at a great enough column position), `indent-relative` either does nothing (if *unindented-ok* is non-`nil`) or calls `tab-to-tab-stop`. Thus, if point is underneath and to the right of the last column of a short line of text, this command ordinarily moves point to the next tab stop by inserting whitespace.

The return value of `indent-relative` is unpredictable.

In the following example, point is at the beginning of the second line:

```
            This line is indented twelve spaces.
    ⋆The quick brown fox jumped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
            This line is indented twelve spaces.
            ⋆The quick brown fox jumped.
```

In this next example, point is between the '`m`' and '`p`' of '`jumped`':

```
            This line is indented twelve spaces.
    The quick brown fox jum⋆ped.
```

Evaluation of the expression `(indent-relative nil)` produces the following:

```
            This line is indented twelve spaces.
    The quick brown fox jum  ⋆ped.
```

`indent-relative-maybe` [Command]
: This command indents the current line like the previous nonblank line, by calling `indent-relative` with `t` as the *unindented-ok* argument. The return value is unpredictable.

  If the previous nonblank line has no indent points beyond the current column, this command does nothing.

### 22.17.5 Adjustable "Tab Stops"

This section explains the mechanism for user-specified "tab stops" and the mechanisms that use and set them. The name "tab stops" is used because the feature is similar to that of the tab stops on a typewriter. The feature works by inserting an appropriate number of spaces and tab characters to reach the next tab stop column; it does not affect the display of tab characters in the buffer (see Section 11.20.1 [Usual Display], page 189). Note that the TAB character as input uses this tab stop feature only in a few major modes, such as Text mode. See Section "Tab Stops" in *The GNU Emacs Manual*.

`tab-to-tab-stop` [Command]
: This command inserts spaces or tabs before point, up to the next tab stop column defined by `tab-stop-list`. It searches the list for an element greater than the current column number, and uses that element as the column to indent to. It does nothing if no such element is found.

`tab-stop-list` [User Option]
: This variable is the list of tab stop columns used by `tab-to-tab-stops`. The elements should be integers in increasing order. The tab stop columns need not be evenly spaced.

  Use `M-x edit-tab-stops` to edit the location of tab stops interactively.

### 22.17.6 Indentation-Based Motion Commands

These commands, primarily for interactive use, act based on the indentation in the text.

`back-to-indentation` [Command]
: This command moves point to the first non-whitespace character in the current line (which is the line in which point is located). It returns `nil`.

`backward-to-indentation` **&optional** *arg* [Command]
: This command moves point backward *arg* lines and then to the first nonblank character on that line. It returns `nil`. If *arg* is omitted or `nil`, it defaults to 1.

`forward-to-indentation` **&optional** *arg* [Command]
: This command moves point forward *arg* lines and then to the first nonblank character on that line. It returns `nil`. If *arg* is omitted or `nil`, it defaults to 1.

## 22.18 Case Changes

The case change commands described here work on text in the current buffer. See ⟨undefined⟩ [Case Conversion], page ⟨undefined⟩, for case conversion functions that work on strings and characters. See ⟨undefined⟩ [Case Tables], page ⟨undefined⟩, for how to customize which characters are upper or lower case and how to convert them.

`capitalize-region` *start end* [Command]

This function capitalizes all words in the region defined by *start* and *end*. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. The function returns `nil`.

If one end of the region is in the middle of a word, the part of the word within the region is treated as an entire word.

When `capitalize-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 44)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

`downcase-region` *start end* [Command]

This function converts all of the letters in the region defined by *start* and *end* to lower case. The function returns `nil`.

When `downcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

`upcase-region` *start end* [Command]

This function converts all of the letters in the region defined by *start* and *end* to upper case. The function returns `nil`.

When `upcase-region` is called interactively, *start* and *end* are point and the mark, with the smallest first.

`capitalize-word` *count* [Command]

This function capitalizes *count* words after point, moving point over as it does. To capitalize means to convert each word's first character to upper case and convert the rest of each word to lower case. If *count* is negative, the function capitalizes the −*count* previous words but does not move point. The value is `nil`.

If point is in the middle of a word, the part of the word before point is ignored when moving forward. The rest is treated as an entire word.

When `capitalize-word` is called interactively, *count* is set to the numeric prefix argument.

`downcase-word` *count* [Command]

This function converts the *count* words after point to all lower case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

When `downcase-word` is called interactively, *count* is set to the numeric prefix argument.

`upcase-word` *count* [Command]

This function converts the *count* words after point to all upper case, moving point over as it does. If *count* is negative, it converts the −*count* previous words but does not move point. The value is `nil`.

When `upcase-word` is called interactively, *count* is set to the numeric prefix argument.

## 22.19 Text Properties

Each character position in a buffer or a string can have a *text property list*, much like the property list of a symbol (see ⟨undefined⟩ [Property Lists], page ⟨undefined⟩). The properties belong to a particular character at a particular place, such as, the letter '`T`' at the beginning of this sentence or the first '`o`' in '`foo`'—if the same character occurs in two different places, the two occurrences in general have different properties.

Each property has a name and a value. Both of these can be any Lisp object, but the name is normally a symbol. Typically each property name symbol is used for a particular purpose; for instance, the text property `face` specifies the faces for displaying the character (see Section 22.19.4 [Special Properties], page 494). The usual way to access the property list is to specify a name and ask what value corresponds to it.

If a character has a `category` property, we call it the *property category* of the character. It should be a symbol. The properties of the symbol serve as defaults for the properties of the character.

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as `substring`, `insert`, and `buffer-substring`.

### 22.19.1 Examining Text Properties

The simplest way to examine text properties is to ask for the value of a particular property of a particular character. For that, use `get-text-property`. Use `text-properties-at` to get the entire property list of a character. See Section 22.19.3 [Property Search], page 492, for functions to examine the properties of a number of characters at once.

These functions handle both strings and buffers. Keep in mind that positions in a string start from 0, whereas positions in a buffer start from 1.

`get-text-property` *pos prop* **&optional** *object* [Function]

This function returns the value of the *prop* property of the character after position *pos* in *object* (a buffer or string). The argument *object* is optional and defaults to the current buffer.

If there is no *prop* property strictly speaking, but the character has a property category that is a symbol, then `get-text-property` returns the *prop* property of that symbol.

`get-char-property` *position prop* **&optional** *object* [Function]

This function is like `get-text-property`, except that it checks overlays first and then text properties. See Section 11.9 [Overlays], page 128.

The argument *object* may be a string, a buffer, or a window. If it is a window, then the buffer displayed in that window is used for text properties and overlays, but only the overlays active for that window are considered. If *object* is a buffer, then overlays

in that buffer are considered first, in order of decreasing priority, followed by the text properties. If *object* is a string, only text properties are considered, since strings never have overlays.

`get-char-property-and-overlay` *position prop* **&optional** *object* [Function]
This is like `get-char-property`, but gives extra information about the overlay that the property value comes from.

Its value is a cons cell whose CAR is the property value, the same value `get-char-property` would return with the same arguments. Its CDR is the overlay in which the property was found, or `nil`, if it was found as a text property or not found at all.

If *position* is at the end of *object*, both the CAR and the CDR of the value are `nil`.

`char-property-alias-alist` [Variable]
This variable holds an alist which maps property names to a list of alternative property names. If a character does not specify a direct value for a property, the alternative property names are consulted in order; the first non-`nil` value is used. This variable takes precedence over `default-text-properties`, and `category` properties take precedence over this variable.

`text-properties-at` *position* **&optional** *object* [Function]
This function returns the entire property list of the character at *position* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

`default-text-properties` [Variable]
This variable holds a property list giving default values for text properties. Whenever a character does not specify a value for a property, neither directly, through a category symbol, or through `char-property-alias-alist`, the value stored in this list is used instead. Here is an example:

```
(setq default-text-properties '(foo 69)
      char-property-alias-alist nil)
;; Make sure character 1 has no properties of its own.
(set-text-properties 1 2 nil)
;; What we get, when we ask, is the default value.
(get-text-property 1 'foo)
     ⇒ 69
```

### 22.19.2 Changing Text Properties

The primitives for changing properties apply to a specified range of text in a buffer or string. The function `set-text-properties` (see end of section) sets the entire property list of the text in that range; more often, it is useful to add, change, or delete just certain properties specified by name.

Since text properties are considered part of the contents of the buffer (or string), and can affect how a buffer looks on the screen, any change in buffer text properties marks the buffer as modified. Buffer text property changes are undoable also (see Section 22.9 [Undo], page 469). Positions in a string start from 0, whereas positions in a buffer start from 1.

`put-text-property` *start end prop value* **&optional** *object* [Function]
This function sets the *prop* property to *value* for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

`add-text-properties` *start end props* **&optional** *object* [Function]
This function adds or overrides text properties for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to add. It should have the form of a property list (see ⟨undefined⟩ [Property Lists], page ⟨undefined⟩): a list whose elements include the property names followed alternately by the corresponding values.

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or its values agree with those in the text).

For example, here is how to set the `comment` and `face` properties of a range of text:

```
(add-text-properties start end
                     '(comment t face highlight))
```

`remove-text-properties` *start end props* **&optional** *object* [Function]
This function deletes specified text properties from the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* specifies which properties to delete. It should have the form of a property list (see ⟨undefined⟩ [Property Lists], page ⟨undefined⟩): a list whose elements are property names alternating with corresponding values. But only the names matter—the values that accompany them are ignored. For example, here's how to remove the `face` property.

```
(remove-text-properties start end '(face nil))
```

The return value is `t` if the function actually changed some property's value; `nil` otherwise (if *props* is `nil` or if no character in the specified text had any of those properties).

To remove all text properties from certain text, use `set-text-properties` and specify `nil` for the new property list.

`remove-list-of-text-properties` *start end list-of-properties* [Function]
**&optional** *object*
Like `remove-text-properties` except that *list-of-properties* is a list of property names only, not an alternating list of property names and values.

`set-text-properties` *start end props* **&optional** *object* [Function]
This function completely replaces the text property list for the text between *start* and *end* in the string or buffer *object*. If *object* is `nil`, it defaults to the current buffer.

The argument *props* is the new property list. It should be a list whose elements are property names alternating with corresponding values.

After `set-text-properties` returns, all the characters in the specified range have identical properties.

If *props* is `nil`, the effect is to get rid of all properties from the specified range of text. Here's an example:

```
(set-text-properties start end nil)
```

Do not rely on the return value of this function.

The easiest way to make a string with text properties is with `propertize`:

`propertize` *string* **&rest** *properties* [Function]

This function returns a copy of *string* which has the text properties *properties*. These properties apply to all the characters in the string that is returned. Here is an example that constructs a string with a `face` property and a `mouse-face` property:

```
(propertize "foo" 'face 'italic
            'mouse-face 'bold-italic)
     ⇒ #("foo" 0 3 (mouse-face bold-italic face italic))
```

To put different properties on various parts of a string, you can construct each part with `propertize` and then combine them with `concat`:

```
(concat
 (propertize "foo" 'face 'italic
             'mouse-face 'bold-italic)
 " and "
 (propertize "bar" 'face 'italic
             'mouse-face 'bold-italic))
     ⇒ #("foo and bar"
                 0 3 (face italic mouse-face bold-italic)
                 3 8 nil
                 8 11 (face italic mouse-face bold-italic))
```

See Section 22.2 [Buffer Contents], page 455, for the function `buffer-substring-no-properties`, which copies text from the buffer but does not copy its properties.

### 22.19.3 Text Property Search Functions

In typical use of text properties, most of the time several or many consecutive characters have the same value for a property. Rather than writing your programs to examine characters one by one, it is much faster to process chunks of text that have the same property value.

Here are functions you can use to do this. They use `eq` for comparing property values. In all cases, *object* defaults to the current buffer.

For good performance, it's very important to use the *limit* argument to these functions, especially the ones that search for a single property—otherwise, they may spend a long time scanning to the end of the buffer, if the property you are interested in does not change.

These functions do not move point; instead, they return a position (or `nil`). Remember that a position is always between two characters; the position returned by these functions is between two characters with different properties.

`next-property-change` *pos* **&optional** *object limit* [Function]

The function scans the text forward from position *pos* in the string or buffer *object* until it finds a change in some text property, then returns the position of the change. In other words, it returns the position of the first character beyond *pos* whose properties are not identical to those of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, this function returns *limit*.

The value is `nil` if the properties remain unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*. The value equals *pos* only when *limit* equals *pos*.

Here is an example of how to scan the buffer by chunks of text within which all properties are constant:

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    Process text from point to next-change...
    (goto-char next-change)))
```

`previous-property-change` *pos* **&optional** *object limit* [Function]
This is like `next-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`next-single-property-change` *pos prop* **&optional** *object limit* [Function]
The function scans text for a change in the *prop* property, then returns the position of the change. The scan goes forward from position *pos* in the string or buffer *object*. In other words, this function returns the position of the first character beyond *pos* whose *prop* property differs from that of the character just after *pos*.

If *limit* is non-`nil`, then the scan ends at position *limit*. If there is no property change before that point, `next-single-property-change` returns *limit*.

The value is `nil` if the property remains unchanged all the way to the end of *object* and *limit* is `nil`. If the value is non-`nil`, it is a position greater than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`previous-single-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-property-change`, but scans back from *pos* instead of forward. If the value is non-`nil`, it is a position less than or equal to *pos*; it equals *pos* only if *limit* equals *pos*.

`next-char-property-change` *pos* **&optional** *limit* [Function]
This is like `next-property-change` except that it considers overlay properties as well as text properties, and if no change is found before the end of the buffer, it returns the maximum buffer position rather than `nil` (in this sense, it resembles the corresponding overlay function `next-overlay-change`, rather than `next-property-change`). There is no *object* operand because this function operates only on the current buffer. It returns the next address at which either kind of property changes.

`previous-char-property-change` *pos* **&optional** *limit* [Function]
This is like `next-char-property-change`, but scans back from *pos* instead of forward, and returns the minimum buffer position if no change is found.

`next-single-char-property-change` *pos prop* **&optional** *object limit* [Function]
This is like `next-single-property-change` except that it considers overlay properties as well as text properties, and if no change is found before the end of the *object*, it returns the maximum valid position in *object* rather than `nil`. Unlike `next-char-property-change`, this function *does* have an *object* operand; if *object* is not a buffer, only text-properties are considered.

`previous-single-char-property-change` *pos prop* **&optional** *object limit* [Function]

This is like `next-single-char-property-change`, but scans back from *pos* instead of forward, and returns the minimum valid position in *object* if no change is found.

`text-property-any` *start end prop value* **&optional** *object* [Function]

This function returns non-`nil` if at least one character between *start* and *end* has a property *prop* whose value is *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

`text-property-not-all` *start end prop value* **&optional** *object* [Function]

This function returns non-`nil` if at least one character between *start* and *end* does not have a property *prop* with value *value*. More precisely, it returns the position of the first such character. Otherwise, it returns `nil`.

The optional fifth argument, *object*, specifies the string or buffer to scan. Positions are relative to *object*. The default for *object* is the current buffer.

### 22.19.4 Properties with Special Meanings

Here is a table of text property names that have special built-in meanings. The following sections list a few additional special property names that control filling and property inheritance. All other names have no standard meaning, and you can use them as you like.

Note: the properties `composition`, `display`, `invisible` and `intangible` can also cause point to move to an acceptable place, after each Emacs command. See Section 2.6 [Adjusting Point], page 23.

`category` If a character has a `category` property, we call it the *property category* of the character. It should be a symbol. The properties of this symbol serve as defaults for the properties of the character.

`face` The `face` property controls the appearance of the character, such as its font and color. See Section 11.12 [Faces], page 137. The value of the property can be the following:

- A face name (a symbol or string).
- A property list of face attributes. This has the form (*keyword value* ...), where each *keyword* is a face attribute name and *value* is a meaningful value for that attribute. With this feature, you do not need to create a face each time you want to specify a particular attribute for certain text.
- A list of faces. This specifies a face which is an aggregate of the attributes of each of the listed faces. Faces occurring earlier in the list have higher priority. Each list element must have one of the two above forms (i.e., either a face name or a property list of face attributes).

Font Lock mode (see Section 20.6 [Font Lock Mode], page 429) works in most buffers by dynamically updating the `face` property of characters based on the context.

`font-lock-face`
: This property specifies a value for the `face` property that Font Lock mode should apply to the underlying text. It is one of the fontification methods used by Font Lock mode, and is useful for special modes that implement their own highlighting. See Section 20.6.6 [Precalculated Fontification], page 436. When Font Lock mode is disabled, `font-lock-face` has no effect.

`mouse-face`
: This property is used instead of `face` when the mouse is on or near the character. For this purpose, "near" means that all text between the character and where the mouse is have the same `mouse-face` property value.

  Emacs ignores all face attributes from the `mouse-face` property that alter the text size (e.g., `:height`, `:weight`, and `:slant`). Those attributes are always the same as for the unhighlighted text.

`fontified`
: This property says whether the text is ready for display. If `nil`, Emacs's redisplay routine calls the functions in `fontification-functions` (see Section 11.12.7 [Auto Faces], page 149) to prepare this part of the buffer before it is displayed. It is used internally by the "just in time" font locking code.

`display`
: This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrow, or replaced with an image. See Section 11.15 [Display Property], page 163.

`help-echo`
: If text has a string as its `help-echo` property, then when you move the mouse onto that text, Emacs displays that string in the echo area, or in the tooltip window (see Section "Tooltips" in *The GNU Emacs Manual*).

  If the value of the `help-echo` property is a function, that function is called with three arguments, *window*, *object* and *pos* and should return a help string or `nil` for none. The first argument, *window* is the window in which the help was found. The second, *object*, is the buffer, overlay or string which had the `help-echo` property. The *pos* argument is as follows:

  - If *object* is a buffer, *pos* is the position in the buffer.
  - If *object* is an overlay, that overlay has a `help-echo` property, and *pos* is the position in the overlay's buffer.
  - If *object* is a string (an overlay string or a string displayed with the `display` property), *pos* is the position in that string.

  If the value of the `help-echo` property is neither a function nor a string, it is evaluated to obtain a help string.

  You can alter the way help text is displayed by setting the variable `show-help-function` (see [Help display], page 499).

  This feature is used in the mode line and for other active text.

`keymap`
: The `keymap` property specifies an additional keymap for commands. When this keymap applies, it is used for key lookup before the minor mode keymaps and

before the buffer's local map. See ⟨undefined⟩ [Active Keymaps], page ⟨undefined⟩. If the property value is a symbol, the symbol's function definition is used as the keymap.

The property's value for the character before point applies if it is non-`nil` and rear-sticky, and the property's value for the character after point applies if it is non-`nil` and front-sticky. (For mouse clicks, the position of the click is used instead of the position of point.)

`local-map`

This property works like `keymap` except that it specifies a keymap to use *instead of* the buffer's local map. For most purposes (perhaps all purposes), it is better to use the `keymap` property.

`syntax-table`

The `syntax-table` property overrides what the syntax table says about this particular character. See ⟨undefined⟩ [Syntax Properties], page ⟨undefined⟩.

`read-only`

If a character has the property `read-only`, then modifying that character is not allowed. Any command that would do so gets an error, `text-read-only`. If the property value is a string, that string is used as the error message.

Insertion next to a read-only character is an error if inserting ordinary text there would inherit the `read-only` property due to stickiness. Thus, you can control permission to insert next to read-only text by controlling the stickiness. See Section 22.19.6 [Sticky Properties], page 500.

Since changing properties counts as modifying the buffer, it is not possible to remove a `read-only` property unless you know the special trick: bind `inhibit-read-only` to a non-`nil` value and then remove the property. See Section 16.7 [Read Only Buffers], page 280.

`invisible`

A non-`nil` `invisible` property can make a character invisible on the screen. See Section 11.6 [Invisible Text], page 121, for details.

`intangible`

If a group of consecutive characters have equal and non-`nil` `intangible` properties, then you cannot place point between them. If you try to move point forward into the group, point actually moves to the end of the group. If you try to move point backward into the group, point actually moves to the start of the group.

If consecutive characters have unequal non-`nil` `intangible` properties, they belong to separate groups; each group is separately treated as described above.

When the variable `inhibit-point-motion-hooks` is non-`nil`, the `intangible` property is ignored.

Beware: this property operates at a very low level, and affects a lot of code in unexpected ways. So use it with extreme caution. A common misuse is to put an intangible property on invisible text, which is actually unnecessary since the command loop will move point outside of the invisible text at the end of each command anyway. See Section 2.6 [Adjusting Point], page 23.

`field` Consecutive characters with the same `field` property constitute a *field*. Some motion functions including `forward-word` and `beginning-of-line` stop moving at a field boundary. See Section 22.19.9 [Fields], page 504.

`cursor` Normally, the cursor is displayed at the beginning or the end of any overlay and text property strings present at the current buffer position. You can place the cursor on any desired character of these strings by giving that character a non-`nil` `cursor` text property. In addition, if the value of the `cursor` property is an integer number, it specifies the number of buffer's character positions, starting with the position where the overlay or the `display` property begins, for which the cursor should be displayed on that character. Specifically, if the value of the `cursor` property of a character is the number *n*, the cursor will be displayed on this character for any buffer position in the range `[`*ovpos*`..`*ovpos*`+`*n*`)`, where *ovpos* is the overlay's starting position given by `overlay-start` (see Section 11.9.1 [Managing Overlays], page 128), or the position where the `display` text property begins in the buffer.

In other words, the string character with the `cursor` property of any non-`nil` value is the character where to display the cursor. The value of the property says for which buffer positions to display the cursor there. If the value is an integer number *n*, the cursor is displayed there when point is anywhere between the beginning of the overlay or `display` property and *n* positions after that. If the value is anything else and non-`nil`, the cursor is displayed there only when point is at the beginning of the `display` property or at `overlay-start`.

When the buffer has many overlay strings (e.g., see Section 11.9.2 [Overlay Properties], page 131) or `display` properties that are strings, it is a good idea to use the `cursor` property on these strings to cue the Emacs display about the places where to put the cursor while traversing these strings. This directly communicates to the display engine where the Lisp program wants to put the cursor, or where the user would expect the cursor.

`pointer` This specifies a specific pointer shape when the mouse pointer is over this text or image. See Section 18.17 [Pointer Shape], page 365, for possible pointer shapes.

`line-spacing`
A newline can have a `line-spacing` text or overlay property that controls the height of the display line ending with that newline. The property value overrides the default frame line spacing and the buffer local `line-spacing` variable. See Section 11.11 [Line Height], page 136.

`line-height`
A newline can have a `line-height` text or overlay property that controls the total height of the display line ending in that newline. See Section 11.11 [Line Height], page 136.

`wrap-prefix`
If text has a `wrap-prefix` property, the prefix it defines will be added at display time to the beginning of every continuation line due to text wrapping (so if lines are truncated, the wrap-prefix is never used). It may be a string or an image (see

Section 11.15.4 [Other Display Specs], page 166), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 11.15.2 [Specified Space], page 164).

A wrap-prefix may also be specified for an entire buffer using the `wrap-prefix` buffer-local variable (however, a `wrap-prefix` text-property takes precedence over the value of the `wrap-prefix` variable). See Section 11.3 [Truncation], page 112.

`line-prefix`

If text has a `line-prefix` property, the prefix it defines will be added at display time to the beginning of every non-continuation line. It may be a string or an image (see Section 11.15.4 [Other Display Specs], page 166), or a stretch of whitespace such as specified by the `:width` or `:align-to` display properties (see Section 11.15.2 [Specified Space], page 164).

A line-prefix may also be specified for an entire buffer using the `line-prefix` buffer-local variable (however, a `line-prefix` text-property takes precedence over the value of the `line-prefix` variable). See Section 11.3 [Truncation], page 112.

`modification-hooks`

If a character has the property `modification-hooks`, then its value should be a list of functions; modifying that character calls all of those functions before the actual modification. Each function receives two arguments: the beginning and end of the part of the buffer being modified. Note that if a particular modification hook function appears on several characters being modified by a single primitive, you can't predict how many times the function will be called. Furthermore, insertion will not modify any existing character, so this hook will only be run when removing some characters, replacing them with others, or changing their text-properties.

If these functions modify the buffer, they should bind `inhibit-modification-hooks` to `t` around doing so, to avoid confusing the internal mechanism that calls these hooks.

Overlays also support the `modification-hooks` property, but the details are somewhat different (see Section 11.9.2 [Overlay Properties], page 131).

`insert-in-front-hooks`
`insert-behind-hooks`

The operation of inserting text in a buffer also calls the functions listed in the `insert-in-front-hooks` property of the following character and in the `insert-behind-hooks` property of the preceding character. These functions receive two arguments, the beginning and end of the inserted text. The functions are called *after* the actual insertion takes place.

See also Section 22.27 [Change Hooks], page 512, for other hooks that are called when you change text in a buffer.

`point-entered`
`point-left`
: The special properties `point-entered` and `point-left` record hook functions that report motion of point. Each time point moves, Emacs compares these two property values:

  - the `point-left` property of the character after the old location, and
  - the `point-entered` property of the character after the new location.

  If these two values differ, each of them is called (if not `nil`) with two arguments: the old value of point, and the new one.

  The same comparison is made for the characters before the old and new locations. The result may be to execute two `point-left` functions (which may be the same function) and/or two `point-entered` functions (which may be the same function). In any case, all the `point-left` functions are called first, followed by all the `point-entered` functions.

  It is possible to use `char-after` to examine characters at various buffer positions without moving point to those positions. Only an actual change in the value of point runs these hook functions.

  The variable `inhibit-point-motion-hooks` can inhibit running the `point-left` and `point-entered` hooks, see [Inhibit point motion hooks], page 499.

`composition`
: This text property is used to display a sequence of characters as a single glyph composed from components. But the value of the property itself is completely internal to Emacs and should not be manipulated directly by, for instance, `put-text-property`.

`inhibit-point-motion-hooks` [Variable]
: When this variable is non-`nil`, `point-left` and `point-entered` hooks are not run, and the `intangible` property has no effect. Do not set this variable globally; bind it with `let`.

`show-help-function` [Variable]
: If this variable is non-`nil`, it specifies a function called to display help strings. These may be `help-echo` properties, menu help strings (see ⟨undefined⟩ [Simple Menu Items], page ⟨undefined⟩, see ⟨undefined⟩ [Extended Menu Items], page ⟨undefined⟩), or tool bar help strings (see ⟨undefined⟩ [Tool Bar], page ⟨undefined⟩). The specified function is called with one argument, the help string to display. Tooltip mode (see Section "Tooltips" in *The GNU Emacs Manual*) provides an example.

### 22.19.5 Formatted Text Properties

These text properties affect the behavior of the fill commands. They are used for representing formatted text. See Section 22.11 [Filling], page 473, and Section 22.12 [Margins], page 475.

`hard`
: If a newline character has this property, it is a "hard" newline. The fill commands do not alter hard newlines and do not move words across them. However, this property takes effect only if the `use-hard-newlines` minor mode is enabled. See Section "Hard and Soft Newlines" in *The GNU Emacs Manual*.

`right-margin`
: This property specifies an extra right margin for filling this part of the text.

`left-margin`
: This property specifies an extra left margin for filling this part of the text.

`justification`
: This property specifies the style of justification for filling this part of the text.

### 22.19.6 Stickiness of Text Properties

Self-inserting characters normally take on the same properties as the preceding character. This is called *inheritance* of properties.

A Lisp program can do insertion with inheritance or without, depending on the choice of insertion primitive. The ordinary text insertion functions, such as `insert`, do not inherit any properties. They insert text with precisely the properties of the string being inserted, and no others. This is correct for programs that copy text from one context to another—for example, into or out of the kill ring. To insert with inheritance, use the special primitives described in this section. Self-inserting characters inherit properties because they work using these primitives.

When you do insertion with inheritance, *which* properties are inherited, and from where, depends on which properties are *sticky*. Insertion after a character inherits those of its properties that are *rear-sticky*. Insertion before a character inherits those of its properties that are *front-sticky*. When both sides offer different sticky values for the same property, the previous character's value takes precedence.

By default, a text property is rear-sticky but not front-sticky; thus, the default is to inherit all the properties of the preceding character, and nothing from the following character.

You can control the stickiness of various text properties with two specific text properties, `front-sticky` and `rear-nonsticky`, and with the variable `text-property-default-nonsticky`. You can use the variable to specify a different default for a given property. You can use those two text properties to make any specific properties sticky or nonsticky in any particular part of the text.

If a character's `front-sticky` property is `t`, then all its properties are front-sticky. If the `front-sticky` property is a list, then the sticky properties of the character are those whose names are in the list. For example, if a character has a `front-sticky` property whose value is `(face read-only)`, then insertion before the character can inherit its `face` property and its `read-only` property, but no others.

The `rear-nonsticky` property works the opposite way. Most properties are rear-sticky by default, so the `rear-nonsticky` property says which properties are *not* rear-sticky. If a character's `rear-nonsticky` property is `t`, then none of its properties are rear-sticky. If the `rear-nonsticky` property is a list, properties are rear-sticky *unless* their names are in the list.

`text-property-default-nonsticky` [Variable]
: This variable holds an alist which defines the default rear-stickiness of various text properties. Each element has the form `(`*`property . nonstickiness`*`)`, and it defines the stickiness of a particular text property, *property*.

If *nonstickiness* is non-`nil`, this means that the property *property* is rear-nonsticky by default. Since all properties are front-nonsticky by default, this makes *property* nonsticky in both directions by default.

The text properties `front-sticky` and `rear-nonsticky`, when used, take precedence over the default *nonstickiness* specified in `text-property-default-nonsticky`.

Here are the functions that insert text with inheritance of properties:

`insert-and-inherit` **&rest** *strings* [Function]
Insert the strings *strings*, just like the function `insert`, but inherit any sticky properties from the adjoining text.

`insert-before-markers-and-inherit` **&rest** *strings* [Function]
Insert the strings *strings*, just like the function `insert-before-markers`, but inherit any sticky properties from the adjoining text.

See Section 22.4 [Insertion], page 458, for the ordinary insertion functions which do not inherit.

### 22.19.7 Lazy Computation of Text Properties

Instead of computing text properties for all the text in the buffer, you can arrange to compute the text properties for parts of the text when and if something depends on them.

The primitive that extracts text from the buffer along with its properties is `buffer-substring`. Before examining the properties, this function runs the abnormal hook `buffer-access-fontify-functions`.

`buffer-access-fontify-functions` [Variable]
This variable holds a list of functions for computing text properties. Before `buffer-substring` copies the text and text properties for a portion of the buffer, it calls all the functions in this list. Each of the functions receives two arguments that specify the range of the buffer being accessed. (The buffer itself is always the current buffer.)

The function `buffer-substring-no-properties` does not call these functions, since it ignores text properties anyway.

In order to prevent the hook functions from being called more than once for the same part of the buffer, you can use the variable `buffer-access-fontified-property`.

`buffer-access-fontified-property` [Variable]
If this variable's value is non-`nil`, it is a symbol which is used as a text property name. A non-`nil` value for that text property means, "the other text properties for this character have already been computed".

If all the characters in the range specified for `buffer-substring` have a non-`nil` value for this property, `buffer-substring` does not call the `buffer-access-fontify-functions` functions. It assumes these characters already have the right text properties, and just copies the properties they already have.

The normal way to use this feature is that the `buffer-access-fontify-functions` functions add this property, as well as others, to the characters they operate on. That way, they avoid being called over and over for the same text.

### 22.19.8 Defining Clickable Text

*Clickable text* is text that can be clicked, with either the mouse or via a keyboard command, to produce some result. Many major modes use clickable text to implement textual hyperlinks, or *links* for short.

The easiest way to insert and manipulate links is to use the `button` package. See Section 11.17 [Buttons], page 179. In this section, we will explain how to manually set up clickable text in a buffer, using text properties. For simplicity, we will refer to the clickable text as a *link*.

Implementing a link involves three separate steps: (1) indicating clickability when the mouse moves over the link; (2) making RET or `Mouse-2` on that link do something; and (3) setting up a `follow-link` condition so that the link obeys `mouse-1-click-follows-link`.

To indicate clickability, add the `mouse-face` text property to the text of the link; then Emacs will highlight the link when the mouse moves over it. In addition, you should define a tooltip or echo area message, using the `help-echo` text property. See Section 22.19.4 [Special Properties], page 494. For instance, here is how Dired indicates that file names are clickable:

```
(if (dired-move-to-filename)
    (add-text-properties
      (point)
      (save-excursion
        (dired-move-to-end-of-filename)
        (point))
      '(mouse-face highlight
        help-echo "mouse-2: visit this file in other window")))
```

To make the link clickable, bind RET and `Mouse-2` to commands that perform the desired action. Each command should check to see whether it was called on a link, and act accordingly. For instance, Dired's major mode keymap binds `Mouse-2` to the following command:

```
(defun dired-mouse-find-file-other-window (event)
  "In Dired, visit the file or directory name you click on."
  (interactive "e")
  (let ((window (posn-window (event-end event)))
        (pos (posn-point (event-end event)))
        file)
    (if (not (windowp window))
        (error "No file chosen"))
    (with-current-buffer (window-buffer window)
      (goto-char pos)
      (setq file (dired-get-file-for-visit)))
    (if (file-directory-p file)
        (or (and (cdr dired-subdir-alist)
                 (dired-goto-subdir file))
            (progn
              (select-window window)
              (dired-other-window file)))
      (select-window window)
      (find-file-other-window (file-name-sans-versions file t)))))
```

This command uses the functions `posn-window` and `posn-point` to determine where the click occurred, and `dired-get-file-for-visit` to determine which file to visit.

Instead of binding the mouse command in a major mode keymap, you can bind it within the link text, using the `keymap` text property (see Section 22.19.4 [Special Properties], page 494). For instance:

```
(let ((map (make-sparse-keymap)))
  (define-key map [mouse-2] 'operate-this-button)
  (put-text-property link-start link-end 'keymap map))
```

With this method, you can easily define different commands for different links. Furthermore, the global definition of RET and `Mouse-2` remain available for the rest of the text in the buffer.

The basic Emacs command for clicking on links is `Mouse-2`. However, for compatibility with other graphical applications, Emacs also recognizes `Mouse-1` clicks on links, provided the user clicks on the link quickly without moving the mouse. This behavior is controlled by the user option `mouse-1-click-follows-link`. See Section "Mouse References" in *The GNU Emacs Manual*.

To set up the link so that it obeys `mouse-1-click-follows-link`, you must either (1) apply a `follow-link` text or overlay property to the link text, or (2) bind the `follow-link` event to a keymap (which can be a major mode keymap or a local keymap specified via the `keymap` text property). The value of the `follow-link` property, or the binding for the `follow-link` event, acts as a "condition" for the link action. This condition tells Emacs two things: the circumstances under which a `Mouse-1` click should be regarded as occurring "inside" the link, and how to compute an "action code" that says what to translate the `Mouse-1` click into. The link action condition can be one of the following:

`mouse-face`
: If the condition is the symbol `mouse-face`, a position is inside a link if there is a non-`nil` `mouse-face` property at that position. The action code is always `t`.

  For example, here is how Info mode handles MOUSE-1:

  ```
  (define-key Info-mode-map [follow-link] 'mouse-face)
  ```

a function
: If the condition is a function, *func*, then a position *pos* is inside a link if `(`*`func pos`*`)` evaluates to non-`nil`. The value returned by *func* serves as the action code.

  For example, here is how pcvs enables `Mouse-1` to follow links on file names only:

  ```
  (define-key map [follow-link]
    (lambda (pos)
      (eq (get-char-property pos 'face) 'cvs-filename-face)))
  ```

anything else
: If the condition value is anything else, then the position is inside a link and the condition itself is the action code. Clearly, you should specify this kind of condition only when applying the condition via a text or property overlay on the link text (so that it does not apply to the entire buffer).

The action code tells `Mouse-1` how to follow the link:

a string or vector
: If the action code is a string or vector, the `Mouse-1` event is translated into the first element of the string or vector; i.e., the action of the `Mouse-1` click is the

local or global binding of that character or symbol. Thus, if the action code is `"foo"`, `Mouse-1` translates into `f`. If it is `[foo]`, `Mouse-1` translates into FOO.

anything else
: For any other non-`nil` action code, the `Mouse-1` event is translated into a `Mouse-2` event at the same position.

To define `Mouse-1` to activate a button defined with `define-button-type`, give the button a `follow-link` property. The property value should be a link action condition, as described above. See Section 11.17 [Buttons], page 179. For example, here is how Help mode handles `Mouse-1`:

```
(define-button-type 'help-xref
  'follow-link t
  'action #'help-button-action)
```

To define `Mouse-1` on a widget defined with `define-widget`, give the widget a `:follow-link` property. The property value should be a link action condition, as described above. For example, here is how the `link` widget specifies that a MOUSE-1 click shall be translated to RET:

```
(define-widget 'link 'item
  "An embedded link."
  :button-prefix 'widget-link-prefix
  :button-suffix 'widget-link-suffix
  :follow-link "\C-m"
  :help-echo "Follow the link."
  :format "%[%t%]")
```

`mouse-on-link-p` *pos* [Function]
: This function returns non-`nil` if position *pos* in the current buffer is on a link. *pos* can also be a mouse event location, as returned by `event-start` (see Section 2.7.13 [Accessing Mouse], page 35).

### 22.19.9 Defining and Using Fields

A field is a range of consecutive characters in the buffer that are identified by having the same value (comparing with `eq`) of the `field` property (either a text-property or an overlay property). This section describes special functions that are available for operating on fields.

You specify a field with a buffer position, *pos*. We think of each field as containing a range of buffer positions, so the position you specify stands for the field containing that position.

When the characters before and after *pos* are part of the same field, there is no doubt which field contains *pos*: the one those characters both belong to. When *pos* is at a boundary between fields, which field it belongs to depends on the stickiness of the `field` properties of the two surrounding characters (see Section 22.19.6 [Sticky Properties], page 500). The field whose property would be inherited by text inserted at *pos* is the field that contains *pos*.

There is an anomalous case where newly inserted text at *pos* would not inherit the `field` property from either side. This happens if the previous character's `field` property is not rear-sticky, and the following character's `field` property is not front-sticky. In this case, *pos* belongs to neither the preceding field nor the following field; the field functions treat it as belonging to an empty field whose beginning and end are both at *pos*.

In all of these functions, if *pos* is omitted or `nil`, the value of point is used by default. If narrowing is in effect, then *pos* should fall within the accessible portion. See ⟨undefined⟩ [Narrowing], page ⟨undefined⟩.

`field-beginning` **&optional** *pos escape-from-edge limit* [Function]
This function returns the beginning of the field specified by *pos*.

If *pos* is at the beginning of its field, and *escape-from-edge* is non-`nil`, then the return value is always the beginning of the preceding field that *ends* at *pos*, regardless of the stickiness of the `field` properties around *pos*.

If *limit* is non-`nil`, it is a buffer position; if the beginning of the field is before *limit*, then *limit* will be returned instead.

`field-end` **&optional** *pos escape-from-edge limit* [Function]
This function returns the end of the field specified by *pos*.

If *pos* is at the end of its field, and *escape-from-edge* is non-`nil`, then the return value is always the end of the following field that *begins* at *pos*, regardless of the stickiness of the `field` properties around *pos*.

If *limit* is non-`nil`, it is a buffer position; if the end of the field is after *limit*, then *limit* will be returned instead.

`field-string` **&optional** *pos* [Function]
This function returns the contents of the field specified by *pos*, as a string.

`field-string-no-properties` **&optional** *pos* [Function]
This function returns the contents of the field specified by *pos*, as a string, discarding text properties.

`delete-field` **&optional** *pos* [Function]
This function deletes the text of the field specified by *pos*.

`constrain-to-field` *new-pos old-pos* **&optional** *escape-from-edge only-in-line inhibit-capture-property* [Function]
This function "constrains" *new-pos* to the field that *old-pos* belongs to—in other words, it returns the position closest to *new-pos* that is in the same field as *old-pos*.

If *new-pos* is `nil`, then `constrain-to-field` uses the value of point instead, and moves point to the resulting position in addition to returning that position.

If *old-pos* is at the boundary of two fields, then the acceptable final positions depend on the argument *escape-from-edge*. If *escape-from-edge* is `nil`, then *new-pos* must be in the field whose `field` property equals what new characters inserted at *old-pos* would inherit. (This depends on the stickiness of the `field` property for the characters before and after *old-pos*.) If *escape-from-edge* is non-`nil`, *new-pos* can be anywhere in the two adjacent fields. Additionally, if two fields are separated by another field with the special value `boundary`, then any point within this special field is also considered to be "on the boundary".

Commands like `C-a` with no argument, that normally move backward to a specific kind of location and stay there once there, probably should specify `nil` for *escape-from-edge*. Other motion commands that check fields should probably pass `t`.

If the optional argument *only-in-line* is non-`nil`, and constraining *new-pos* in the usual way would move it to a different line, *new-pos* is returned unconstrained. This used in commands that move by line, such as `next-line` and `beginning-of-line`, so that they respect field boundaries only in the case where they can still move to the right line.

If the optional argument *inhibit-capture-property* is non-`nil`, and *old-pos* has a non-`nil` property of that name, then any field boundaries are ignored.

You can cause `constrain-to-field` to ignore all field boundaries (and so never constrain anything) by binding the variable `inhibit-field-text-motion` to a non-`nil` value.

### 22.19.10 Why Text Properties are not Intervals

Some editors that support adding attributes to text in the buffer do so by letting the user specify "intervals" within the text, and adding the properties to the intervals. Those editors permit the user or the programmer to determine where individual intervals start and end. We deliberately provided a different sort of interface in Emacs Lisp to avoid certain paradoxical behavior associated with text modification.

If the actual subdivision into intervals is meaningful, that means you can distinguish between a buffer that is just one interval with a certain property, and a buffer containing the same text subdivided into two intervals, both of which have that property.

Suppose you take the buffer with just one interval and kill part of the text. The text remaining in the buffer is one interval, and the copy in the kill ring (and the undo list) becomes a separate interval. Then if you yank back the killed text, you get two intervals with the same properties. Thus, editing does not preserve the distinction between one interval and two.

Suppose we "fix" this problem by coalescing the two intervals when the text is inserted. That works fine if the buffer originally was a single interval. But suppose instead that we have two adjacent intervals with the same properties, and we kill the text of one interval and yank it back. The same interval-coalescence feature that rescues the other case causes trouble in this one: after yanking, we have just one interval. One again, editing does not preserve the distinction between one interval and two.

Insertion of text at the border between intervals also raises questions that have no satisfactory answer.

However, it is easy to arrange for editing to behave consistently for questions of the form, "What are the properties of this character?" So we have decided these are the only questions that make sense; we have not implemented asking questions about where intervals start or end.

In practice, you can usually use the text property search functions in place of explicit interval boundaries. You can think of them as finding the boundaries of intervals, assuming that intervals are always coalesced whenever possible. See Section 22.19.3 [Property Search], page 492.

Emacs also provides explicit intervals as a presentation feature; see Section 11.9 [Overlays], page 128.

## 22.20 Substituting for a Character Code

The following functions replace characters within a specified region based on their character codes.

`subst-char-in-region` *start end old-char new-char* **&optional** *noundo* [Function]
This function replaces all occurrences of the character *old-char* with the character *new-char* in the region of the current buffer defined by *start* and *end*.

If *noundo* is non-`nil`, then `subst-char-in-region` does not record the change for undo and does not mark the buffer as modified. This was useful for controlling the old selective display feature (see Section 11.7 [Selective Display], page 124).

`subst-char-in-region` does not move point and returns `nil`.

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

`translate-region` *start end table* [Command]
This function applies a translation table to the characters in the buffer between positions *start* and *end*.

The translation table *table* is a string or a char-table; `(aref table ochar)` gives the translated character corresponding to *ochar*. If *table* is a string, any characters with codes larger than the length of *table* are not altered by the translation.

The return value of `translate-region` is the number of characters that were actually changed by the translation. This does not count characters that were mapped into themselves in the translation table.

## 22.21 Registers

A register is a sort of variable used in Emacs editing that can hold a variety of different kinds of values. Each register is named by a single character. All ASCII characters and their meta variants (but with the exception of `C-g`) can be used to name registers. Thus, there are 255 possible registers. A register is designated in Emacs Lisp by the character that is its name.

`register-alist` [Variable]
This variable is an alist of elements of the form `(name . contents)`. Normally, there is one element for each Emacs register that has been used.

The object *name* is a character (an integer) identifying the register.

The *contents* of a register can have several possible types:

a number
: A number stands for itself. If `insert-register` finds a number in the register, it converts the number to decimal.

a marker
: A marker represents a buffer position to jump to.

a string
: A string is text saved in the register.

a rectangle
: A rectangle is represented by a list of strings.

`(`*`window-configuration position`*`)`
: This represents a window configuration to restore in one frame, and a position to jump to in the current buffer.

`(`*`frame-configuration position`*`)`
: This represents a frame configuration to restore, and a position to jump to in the current buffer.

(file *filename*)
: This represents a file to visit; jumping to this value visits file *filename*.

(file-query *filename position*)
: This represents a file to visit and a position in it; jumping to this value visits file *filename* and goes to buffer position *position*. Restoring this type of position asks the user for confirmation first.

The functions in this section return unpredictable values unless otherwise stated.

`get-register` *reg* [Function]
: This function returns the contents of the register *reg*, or `nil` if it has no contents.

`set-register` *reg value* [Function]
: This function sets the contents of register *reg* to *value*. A register can be set to any value, but the other register functions expect only certain data types. The return value is *value*.

`view-register` *reg* [Command]
: This command displays what is contained in register *reg*.

`insert-register` *reg* **&optional** *beforep* [Command]
: This command inserts contents of register *reg* into the current buffer.

  Normally, this command puts point before the inserted text, and the mark after it. However, if the optional second argument *beforep* is non-`nil`, it puts the mark before and point after. You can pass a non-`nil` second argument *beforep* to this function interactively by supplying any prefix argument.

  If the register contains a rectangle, then the rectangle is inserted with its upper left corner at point. This means that text is inserted in the current line and underneath it on successive lines.

  If the register contains something other than saved text (a string) or a rectangle (a list), currently useless things happen. This may be changed in the future.

## 22.22 Transposition of Text

This function can be used to transpose stretches of text:

`transpose-regions` *start1 end1 start2 end2* **&optional** *leave-markers* [Function]
: This function exchanges two nonoverlapping portions of the buffer. Arguments *start1* and *end1* specify the bounds of one portion and arguments *start2* and *end2* specify the bounds of the other portion.

  Normally, `transpose-regions` relocates markers with the transposed text; a marker previously positioned within one of the two transposed portions moves along with that portion, thus remaining between the same two characters in their new position. However, if *leave-markers* is non-`nil`, `transpose-regions` does not do this—it leaves all markers unrelocated.

## 22.23 Base 64 Encoding

Base 64 code is used in email to encode a sequence of 8-bit bytes as a longer sequence of ASCII graphic characters. It is defined in Internet RFC[1]2045. This section describes the functions for converting to and from this code.

`base64-encode-region` *beg end* **&optional** *no-line-break* [Command]
: This function converts the region from *beg* to *end* into base 64 code. It returns the length of the encoded text. An error is signaled if a character in the region is multibyte, i.e., in a multibyte buffer the region must contain only characters from the charsets `ascii`, `eight-bit-control` and `eight-bit-graphic`.

  Normally, this function inserts newline characters into the encoded text, to avoid overlong lines. However, if the optional argument *no-line-break* is non-`nil`, these newlines are not added, so the output is just one long line.

`base64-encode-string` *string* **&optional** *no-line-break* [Function]
: This function converts the string *string* into base 64 code. It returns a string containing the encoded text. As for `base64-encode-region`, an error is signaled if a character in the string is multibyte.

  Normally, this function inserts newline characters into the encoded text, to avoid overlong lines. However, if the optional argument *no-line-break* is non-`nil`, these newlines are not added, so the result string is just one long line.

`base64-decode-region` *beg end* [Command]
: This function converts the region from *beg* to *end* from base 64 code into the corresponding decoded text. It returns the length of the decoded text.

  The decoding functions ignore newline characters in the encoded text.

`base64-decode-string` *string* [Function]
: This function converts the string *string* from base 64 code into the corresponding decoded text. It returns a unibyte string containing the decoded text.

  The decoding functions ignore newline characters in the encoded text.

[1] An RFC, an acronym for *Request for Comments*, is a numbered Internet informational document describing a standard. RFCs are usually written by technical experts acting on their own initiative, and are traditionally written in a pragmatic, experience-driven manner.

## 22.24 Checksum/Hash

Emacs has built-in support for computing *cryptographic hashes*. A cryptographic hash, or *checksum*, is a digital "fingerprint" of a piece of data (e.g., a block of text) which can be used to check that you have an unaltered copy of that data.

Emacs supports several common cryptographic hash algorithms: MD5, SHA-1, SHA-2, SHA-224, SHA-256, SHA-384 and SHA-512. MD5 is the oldest of these algorithms, and is commonly used in *message digests* to check the integrity of messages transmitted over a network. MD5 is not "collision resistant" (i.e., it is possible to deliberately design different pieces of data which have the same MD5 hash), so you should not used it for anything security-related. A similar theoretical weakness also exists in SHA-1. Therefore, for security-related applications you should use the other hash types, such as SHA-2.

`secure-hash` *algorithm object* **&optional** *start end binary* [Function]

This function returns a hash for *object*. The argument *algorithm* is a symbol stating which hash to compute: one of `md5`, `sha1`, `sha224`, `sha256`, `sha384` or `sha512`. The argument *object* should be a buffer or a string.

The optional arguments *start* and *end* are character positions specifying the portion of *object* to compute the message digest for. If they are `nil` or omitted, the hash is computed for the whole of *object*.

If the argument *binary* is omitted or `nil`, the function returns the *text form* of the hash, as an ordinary Lisp string. If *binary* is non-`nil`, it returns the hash in *binary form*, as a sequence of bytes stored in a unibyte string.

This function does not compute the hash directly from the internal representation of *object*'s text (see ⟨undefined⟩ [Text Representations], page ⟨undefined⟩). Instead, it encodes the text using a coding system (see Section 19.6 [Coding Systems], page 381), and computes the hash from that encoded text. If *object* is a buffer, the coding system used is the one which would be chosen by default for writing the text into a file. If *object* is a string, the user's preferred coding system is used (see Section "Recognize Coding" in *GNU Emacs Manual*).

`md5` *object* **&optional** *start end coding-system noerror* [Function]

This function returns an MD5 hash. It is semi-obsolete, since for most purposes it is equivalent to calling `secure-hash` with `md5` as the *algorithm* argument. The *object*, *start* and *end* arguments have the same meanings as in `secure-hash`.

If *coding-system* is non-`nil`, it specifies a coding system to use to encode the text; if omitted or `nil`, the default coding system is used, like in `secure-hash`.

Normally, `md5` signals an error if the text can't be encoded using the specified or chosen coding system. However, if *noerror* is non-`nil`, it silently uses `raw-text` coding instead.

## 22.25 Parsing HTML and XML

When Emacs is compiled with libxml2 support, the following functions are available to parse HTML or XML text into Lisp object trees.

`libxml-parse-html-region` *start end* **&optional** *base-url* [Function]

This function parses the text between *start* and *end* as HTML, and returns a list representing the HTML *parse tree*. It attempts to handle "real world" HTML by robustly coping with syntax mistakes.

The optional argument *base-url*, if non-`nil`, should be a string specifying the base URL for relative URLs occurring in links.

In the parse tree, each HTML node is represented by a list in which the first element is a symbol representing the node name, the second element is an alist of node attributes, and the remaining elements are the subnodes.

The following example demonstrates this. Given this (malformed) HTML document:

```
<html><head></head><body width=101><div class=thing>Foo<div>Yes
```

A call to `libxml-parse-html-region` returns this:

```
(html ()
  (head ())
  (body ((width . "101"))
   (div ((class . "thing"))
    "Foo"
    (div ()
      "Yes"))))
```

`libxml-parse-xml-region` *start end* **&optional** *base-url* [Function]

This function is the same as `libxml-parse-html-region`, except that it parses the text as XML rather than HTML (so it is stricter about syntax).

## 22.26 Atomic Change Groups

In database terminology, an *atomic* change is an indivisible change—it can succeed entirely or it can fail entirely, but it cannot partly succeed. A Lisp program can make a series of changes to one or several buffers as an *atomic change group*, meaning that either the entire series of changes will be installed in their buffers or, in case of an error, none of them will be.

To do this for one buffer, the one already current, simply write a call to `atomic-change-group` around the code that makes the changes, like this:

```
(atomic-change-group
  (insert foo)
  (delete-region x y))
```

If an error (or other nonlocal exit) occurs inside the body of `atomic-change-group`, it unmakes all the changes in that buffer that were during the execution of the body. This kind of change group has no effect on any other buffers—any such changes remain.

If you need something more sophisticated, such as to make changes in various buffers constitute one atomic group, you must directly call lower-level functions that `atomic-change-group` uses.

`prepare-change-group` **&optional** *buffer* [Function]

This function sets up a change group for buffer *buffer*, which defaults to the current buffer. It returns a "handle" that represents the change group. You must use this handle to activate the change group and subsequently to finish it.

To use the change group, you must *activate* it. You must do this before making any changes in the text of *buffer*.

`activate-change-group` *handle* [Function]
: This function activates the change group that *handle* designates.

After you activate the change group, any changes you make in that buffer become part of it. Once you have made all the desired changes in the buffer, you must *finish* the change group. There are two ways to do this: you can either accept (and finalize) all the changes, or cancel them all.

`accept-change-group` *handle* [Function]
: This function accepts all the changes in the change group specified by *handle*, making them final.

`cancel-change-group` *handle* [Function]
: This function cancels and undoes all the changes in the change group specified by *handle*.

Your code should use `unwind-protect` to make sure the group is always finished. The call to `activate-change-group` should be inside the `unwind-protect`, in case the user types `C-g` just after it runs. (This is one reason why `prepare-change-group` and `activate-change-group` are separate functions, because normally you would call `prepare-change-group` before the start of that `unwind-protect`.) Once you finish the group, don't use the handle again—in particular, don't try to finish the same group twice.

To make a multibuffer change group, call `prepare-change-group` once for each buffer you want to cover, then use `nconc` to combine the returned values, like this:

```
(nconc (prepare-change-group buffer-1)
       (prepare-change-group buffer-2))
```

You can then activate the multibuffer change group with a single call to `activate-change-group`, and finish it with a single call to `accept-change-group` or `cancel-change-group`.

Nested use of several change groups for the same buffer works as you would expect. Non-nested use of change groups for the same buffer will get Emacs confused, so don't let it happen; the first change group you start for any given buffer should be the last one finished.

## 22.27 Change Hooks

These hook variables let you arrange to take notice of all changes in all buffers (or in a particular buffer, if you make them buffer-local). See also Section 22.19.4 [Special Properties], page 494, for how to detect changes to specific parts of the text.

The functions you use in these hooks should save and restore the match data if they do anything that uses regular expressions; otherwise, they will interfere in bizarre ways with the editing operations that call them.

`before-change-functions` [Variable]
: This variable holds a list of functions to call before any buffer modification. Each function gets two arguments, the beginning and end of the region that is about to change, represented as integers. The buffer that is about to change is always the current buffer.

`after-change-functions` [Variable]

This variable holds a list of functions to call after any buffer modification. Each function receives three arguments: the beginning and end of the region just changed, and the length of the text that existed before the change. All three arguments are integers. The buffer that has been changed is always the current buffer.

The length of the old text is the difference between the buffer positions before and after that text as it was before the change. As for the changed text, its length is simply the difference between the first two arguments.

Output of messages into the '`*Messages*`' buffer does not call these functions.

`combine-after-change-calls` *body...* [Macro]

The macro executes *body* normally, but arranges to call the after-change functions just once for a series of several changes—if that seems safe.

If a program makes several text changes in the same area of the buffer, using the macro `combine-after-change-calls` around that part of the program can make it run considerably faster when after-change hooks are in use. When the after-change hooks are ultimately called, the arguments specify a portion of the buffer including all of the changes made within the `combine-after-change-calls` body.

**Warning:** You must not alter the values of `after-change-functions` within the body of a `combine-after-change-calls` form.

**Warning:** if the changes you combine occur in widely scattered parts of the buffer, this will still work, but it is not advisable, because it may lead to inefficient behavior for some change hook functions.

`first-change-hook` [Variable]

This variable is a normal hook that is run whenever a buffer is changed that was previously in the unmodified state.

`inhibit-modification-hooks` [Variable]

If this variable is non-`nil`, all of the change hooks are disabled; none of them run. This affects all the hook variables described above in this section, as well as the hooks attached to certain special text properties (see Section 22.19.4 [Special Properties], page 494) and overlay properties (see Section 11.9.2 [Overlay Properties], page 131).

Also, this variable is bound to non-`nil` while running those same hook variables, so that by default modifying the buffer from a modification hook does not cause other modification hooks to be run. If you do want modification hooks to be run in a particular piece of code that is itself run from a modification hook, then rebind locally `inhibit-modification-hooks` to `nil`.

# 23 Editing Programs

This chapter describes Emacs features for facilitating editing programs. Some of the things these features can do are:

- Find or move over top-level definitions (see Section 23.2 [Defuns], page 515).
- Apply the usual indentation conventions of the language (see Section 23.3 [Program Indent], page 517).
- Balance parentheses (see Section 23.4 [Parentheses], page 520).
- Insert, kill or align comments (see Section 23.5 [Comments], page 523).
- Highlight program syntax (see ⟨undefined⟩ [Font Lock], page ⟨undefined⟩).

## 23.1 Major Modes for Programming Languages

Emacs has specialized major modes (see Section 20.2 [Major Modes], page 399) for many programming languages. A programming language mode typically specifies the syntax of expressions, the customary rules for indentation, how to do syntax highlighting for the language, and how to find the beginning or end of a function definition. It often has features for compiling and debugging programs as well. The major mode for each language is named after the language; for instance, the major mode for the C programming language is `c-mode`.

Emacs has programming language modes for Lisp, Scheme, the Scheme-based DSSSL expression language, Ada, ASM, AWK, C, C++, Delphi, Fortran, Icon, IDL (CORBA), IDLWAVE, Java, Javascript, Metafont (TEX's companion for font creation), Modula2, Objective-C, Octave, Pascal, Perl, Pike, PostScript, Prolog, Python, Ruby, Simula, Tcl, and VHDL. An alternative mode for Perl is called CPerl mode. Modes are also available for the scripting languages of the common GNU and Unix shells, VMS DCL, and MS-DOS/MS-Windows '`BAT`' files, and for makefiles, DNS master files, and various sorts of configuration files.

Ideally, Emacs should have a major mode for each programming language that you might want to edit. If it doesn't have a mode for your favorite language, the mode might be implemented in a package not distributed with Emacs (see ⟨undefined⟩ [Packages], page ⟨undefined⟩); or you can contribute one.

In most programming languages, indentation should vary from line to line to illustrate the structure of the program. Therefore, in most programming language modes, typing TAB updates the indentation of the current line (see Section 23.3 [Program Indent], page 517). Furthermore, DEL is usually bound to `backward-delete-char-untabify`, which deletes backward treating each tab as if it were the equivalent number of spaces, so that you can delete one column of indentation without worrying whether the whitespace consists of spaces or tabs.

Entering a programming language mode runs the custom Lisp functions specified in the hook variable `prog-mode-hook`, followed by those specified in the mode's own mode hook (see Section 20.2 [Major Modes], page 399). For instance, entering C mode runs the hooks `prog-mode-hook` and `c-mode-hook`. See Section 33.2.2 [Hooks], page 696, for information about hooks.

The Emacs distribution contains Info manuals for the major modes for Ada, C/C++/Objective C/Java/Corba IDL/Pike/AWK, and IDLWAVE. For Fortran mode, see Section "Fortran" in *Specialized Emacs Features*.

## 23.2 Top-Level Definitions, or Defuns

In Emacs, a major definition at the top level in the buffer, such as a function, is called a *defun*. The name comes from Lisp, but in Emacs we use it for all languages.

### 23.2.1 Left Margin Convention

Many programming-language modes assume by default that any opening delimiter found at the left margin is the start of a top-level definition, or defun. Therefore, **don't put an opening delimiter at the left margin unless it should have that significance**. For instance, never put an open-parenthesis at the left margin in a Lisp file unless it is the start of a top-level list.

The convention speeds up many Emacs operations, which would otherwise have to scan back to the beginning of the buffer to analyze the syntax of the code.

If you don't follow this convention, not only will you have trouble when you explicitly use the commands for motion by defuns; other features that use them will also give you trouble. This includes the indentation commands (see Section 23.3 [Program Indent], page 517) and Font Lock mode (see ⟨undefined⟩ [Font Lock], page ⟨undefined⟩).

The most likely problem case is when you want an opening delimiter at the start of a line inside a string. To avoid trouble, put an escape character ('\', in C and Emacs Lisp, '/' in some other Lisp dialects) before the opening delimiter. This will not affect the contents of the string, but will prevent that opening delimiter from starting a defun. Here's an example:

```
  (insert "Foo:
\(bar)
")
```

To help you catch violations of this convention, Font Lock mode highlights confusing opening delimiters (those that ought to be quoted) in bold red.

If you need to override this convention, you can do so by setting the variable `open-paren-in-column-0-is-defun-start`. If this user option is set to `t` (the default), opening parentheses or braces at column zero always start defuns. When it is `nil`, defuns are found by searching for parens or braces at the outermost level.

Usually, you should leave this option at its default value of `t`. If your buffer contains parentheses or braces in column zero which don't start defuns, and it is somehow impractical to remove these parentheses or braces, it might be helpful to set the option to `nil`. Be aware that this might make scrolling and display in large buffers quite sluggish. Furthermore, the parentheses and braces must be correctly matched throughout the buffer for it to work properly.

### 23.2.2 Moving by Defuns

These commands move point or set up the region based on top-level major definitions, also called *defuns*.

`C-M-a` Move to beginning of current or preceding defun (`beginning-of-defun`).

`C-M-e` Move to end of current or following defun (`end-of-defun`).

`C-M-h` Put region around whole current or following defun (`mark-defun`).

The commands to move to the beginning and end of the current defun are `C-M-a` (`beginning-of-defun`) and `C-M-e` (`end-of-defun`). If you repeat one of these commands, or use a positive numeric argument, each repetition moves to the next defun in the direction of motion.

`C-M-a` with a negative argument $-n$ moves forward $n$ times to the next beginning of a defun. This is not exactly the same place that `C-M-e` with argument $n$ would move to; the end of this defun is not usually exactly the same place as the beginning of the following defun. (Whitespace, comments, and perhaps declarations can separate them.) Likewise, `C-M-e` with a negative argument moves back to an end of a defun, which is not quite the same as `C-M-a` with a positive argument.

To operate on the current defun, use `C-M-h` (`mark-defun`), which sets the mark at the end of the current defun and puts point at its beginning. See Section 8.2 [Marking Objects], page 90. This is the easiest way to get ready to kill the defun in order to move it to a different place in the file. If you use the command while point is between defuns, it uses the following defun. If you use the command while the mark is already active, it sets the mark but does not move point; furthermore, each successive use of `C-M-h` extends the end of the region to include one more defun.

In C mode, `C-M-h` runs the function `c-mark-function`, which is almost the same as `mark-defun`; the difference is that it backs up over the argument declarations, function name and returned data type so that the entire C function is inside the region. This is an example of how major modes adjust the standard key bindings so that they do their standard jobs in a way better fitting a particular language. Other major modes may replace any or all of these key bindings for that purpose.

### 23.2.3 Imenu

The Imenu facility offers a way to find the major definitions in a file by name. It is also useful in text formatter major modes, where it treats each chapter, section, etc., as a definition. (See Section 25.3 [Tags], page 571, for a more powerful feature that handles multiple files together.)

If you type `M-x imenu`, it reads the name of a definition using the minibuffer, then moves point to that definition. You can use completion to specify the name; the command always displays the whole list of valid names.

Alternatively, you can bind the command `imenu` to a mouse click. Then it displays mouse menus for you to select a definition name. You can also add the buffer's index to the menu bar by calling `imenu-add-menubar-index`. If you want to have this menu bar item available for all buffers in a certain major mode, you can do this by adding `imenu-add-menubar-index` to its mode hook. But if you have done that, you will have to wait a little while each time you visit a file in that mode, while Emacs finds all the definitions in that buffer.

When you change the contents of a buffer, if you add or delete definitions, you can update the buffer's index based on the new contents by invoking the '`*Rescan*`' item in the menu. Rescanning happens automatically if you set `imenu-auto-rescan` to a non-`nil` value. There is no need to rescan because of small changes in the text.

You can customize the way the menus are sorted by setting the variable `imenu-sort-function`. By default, names are ordered as they occur in the buffer; if you want alphabetic sorting, use the symbol `imenu--sort-by-name` as the value. You can also define your own comparison function by writing Lisp code.

Imenu provides the information to guide Which Function mode (see below). The Speedbar can also use it (see ⟨undefined⟩ [Speedbar], page ⟨undefined⟩).

### 23.2.4 Which Function Mode

Which Function mode is a global minor mode (see Section 20.3 [Minor Modes], page 413) which displays the current function name in the mode line, updating it as you move around in a buffer.

To either enable or disable Which Function mode, use the command `M-x which-function-mode`. Which Function mode is a global minor mode. By default, it takes effect in all major modes major modes that know how to support it (i.e., all the major modes that support Imenu). You can restrict it to a specific list of major modes by changing the value of the variable `which-func-modes` from `t` (which means to support all available major modes) to a list of major mode names.

## 23.3 Indentation for Programs

The best way to keep a program properly indented is to use Emacs to reindent it as you change it. Emacs has commands to indent either a single line, a specified number of lines, or all of the lines inside a single parenthetical grouping.

See Section 22.17 [Indentation], page 483, for general information about indentation. This section describes indentation features specific to programming language modes.

Emacs also provides a Lisp pretty-printer in the `pp` package, which reformats Lisp objects with nice-looking indentation.

### 23.3.1 Basic Program Indentation Commands

TAB Adjust indentation of current line (`indent-for-tab-command`).

`C-j` Insert a newline, then adjust indentation of following line (`newline-and-indent`).

The basic indentation command is TAB (`indent-for-tab-command`), which was documented in Section 22.17 [Indentation], page 483. In programming language modes, TAB indents the current line, based on the indentation and syntactic content of the preceding lines; if the region is active, TAB indents each line within the region, not just the current line.

The command `C-j` (`newline-and-indent`), which was documented in Section 21.1 [Indentation Commands], page 451, does the same as RET followed by TAB: it inserts a new line, then adjusts the line's indentation.

When indenting a line that starts within a parenthetical grouping, Emacs usually places the start of the line under the preceding line within the group, or under the text after the parenthesis. If you manually give one of these lines a nonstandard indentation (e.g., for aesthetic purposes), the lines below will follow it.

The indentation commands for most programming language modes assume that a open-parenthesis, open-brace or other opening delimiter at the left margin is the start of a function. If the code you are editing violates this assumption—even if the delimiters occur in strings or comments—you must set `open-paren-in-column-0-is-defun-start` to `nil` for indentation to work properly. See Section 23.2.1 [Left Margin Paren], page 515.

### 23.3.2 Indenting Several Lines

Sometimes, you may want to reindent several lines of code at a time. One way to do this is to use the mark; when the mark is active and the region is non-empty, TAB indents every line in the region. Alternatively, the command `C-M-\` (`indent-region`) indents every line in the region, whether or not the mark is active (see Section 21.1 [Indentation Commands], page 451).

In addition, Emacs provides the following commands for indenting large chunks of code:

`C-M-q` Reindent all the lines within one parenthetical grouping.

`C-u TAB` Shift an entire parenthetical grouping rigidly sideways so that its first line is properly indented.

`M-x indent-code-rigidly`
Shift all the lines in the region rigidly sideways, but do not alter lines that start inside comments and strings.

To reindent the contents of a single parenthetical grouping, position point before the beginning of the grouping and type `C-M-q`. This changes the relative indentation within the grouping, without affecting its overall indentation (i.e., the indentation of the line where the grouping starts). The function that `C-M-q` runs depends on the major mode; it is `indent-pp-sexp` in Lisp mode, `c-indent-exp` in C mode, etc. To correct the overall indentation as well, type TAB first.

If you like the relative indentation within a grouping but not the indentation of its first line, move point to that first line and type `C-u TAB`. In Lisp, C, and some other major modes, TAB with a numeric argument reindents the current line as usual, then reindents by the same amount all the lines in the parenthetical grouping starting on the current line. It is clever, though, and does not alter lines that start inside strings. Neither does it alter C preprocessor lines when in C mode, but it does reindent any continuation lines that may be attached to them.

The command `M-x indent-code-rigidly` rigidly shifts all the lines in the region sideways, like `indent-rigidly` does (see Section 21.1 [Indentation Commands], page 451). It doesn't alter the indentation of lines that start inside a string, unless the region also starts inside that string. The prefix arg specifies the number of columns to indent.

### 23.3.3 Customizing Lisp Indentation

The indentation pattern for a Lisp expression can depend on the function called by the expression. For each Lisp function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a Lisp program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the

expression; otherwise, the second line is indented underneath the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable `lisp-indent-offset` is non-`nil`, it overrides the usual indentation pattern for the second line of an expression, so that such lines are always indented `lisp-indent-offset` more columns than the containing list.

Certain functions override the standard pattern. Functions whose names start with `def` treat the second lines as the start of a *body*, by indenting the second line `lisp-body-indent` additional columns beyond the open-parenthesis that starts the expression.

You can override the standard pattern in various ways for individual functions, according to the `lisp-indent-function` property of the function name. This is normally done for macro definitions, using the `declare` construct. See Section "Defining Macros" in *the Emacs Lisp Reference Manual*.

### 23.3.4 Commands for C Indentation

Here are special features for indentation in C mode and related modes:

`C-c C-q` Reindent the current top-level function definition or aggregate type declaration (`c-indent-defun`).

`C-M-q` Reindent each line in the balanced expression that follows point (`c-indent-exp`). A prefix argument inhibits warning messages about invalid syntax.

`TAB` Reindent the current line, and/or in some cases insert a tab character (`c-indent-command`).

If `c-tab-always-indent` is `t`, this command always reindents the current line and does nothing else. This is the default.

If that variable is `nil`, this command reindents the current line only if point is at the left margin or in the line's indentation; otherwise, it inserts a tab (or the equivalent number of spaces, if `indent-tabs-mode` is `nil`).

Any other value (not `nil` or `t`) means always reindent the line, and also insert a tab if within a comment or a string.

To reindent the whole current buffer, type `C-x h C-M-\`. This first selects the whole buffer as the region, then reindents that region.

To reindent the current block, use `C-M-u C-M-q`. This moves to the front of the block and then reindents it all.

### 23.3.5 Customizing C Indentation

C mode and related modes use a flexible mechanism for customizing indentation. C mode indents a source line in two steps: first it classifies the line syntactically according to its contents and context; second, it determines the indentation offset associated by your selected *style* with the syntactic construct and adds this onto the indentation of the *anchor statement*.

`C-c . RET` *style* `RET`
Select a predefined style *style* (`c-set-style`).

A *style* is a named collection of customizations that can be used in C mode and the related modes. Section "Styles" in *The CC Mode Manual*, for a complete description.

Emacs comes with several predefined styles, including `gnu`, `k&r`, `bsd`, `stroustrup`, `linux`, `python`, `java`, `whitesmith`, `ellemtel`, and `awk`. Some of these styles are primarily intended for one language, but any of them can be used with any of the languages supported by these modes. To find out what a style looks like, select it and reindent some code, e.g., by typing C-M-Q at the start of a function definition.

To choose a style for the current buffer, use the command `C-c .`. Specify a style name as an argument (case is not significant). This command affects the current buffer only, and it affects only future invocations of the indentation commands; it does not reindent the code already in the buffer. To reindent the whole buffer in the new style, you can type `C-x h C-M-\`.

You can also set the variable `c-default-style` to specify the default style for various major modes. Its value should be either the style's name (a string) or an alist, in which each element specifies one major mode and which indentation style to use for it. For example,

```
(setq c-default-style
      '((java-mode . "java")
        (awk-mode . "awk")
        (other . "gnu")))
```

specifies explicit choices for Java and AWK modes, and the default '`gnu`' style for the other C-like modes. (These settings are actually the defaults.) This variable takes effect when you select one of the C-like major modes; thus, if you specify a new default style for Java mode, you can make it take effect in an existing Java mode buffer by typing `M-x java-mode` there.

The `gnu` style specifies the formatting recommended by the GNU Project for C; it is the default, so as to encourage use of our recommended style.

See Section "Indentation Engine Basics" in *the CC Mode Manual*, and Section "Customizing Indentation" in *the CC Mode Manual*, for more information on customizing indentation for C and related modes, including how to override parts of an existing style and how to define your own styles.

As an alternative to specifying a style, you can tell Emacs to guess a style by typing `M-x c-guess` in a sample code buffer. You can then apply the guessed style to other buffers with `M-x c-guess-install`. See Section "Guessing the Style" in *the CC Mode Manual*, for details.

## 23.4 Commands for Editing with Parentheses

This section describes the commands and features that take advantage of the parenthesis structure in a program, or help you keep it balanced.

When talking about these facilities, the term "parenthesis" also includes braces, brackets, or whatever delimiters are defined to match in pairs. The major mode controls which delimiters are significant, through the syntax table (see Section "Syntax Tables" in *The Emacs Lisp Reference Manual*). In Lisp, only parentheses count; in C, these commands apply to braces and brackets too.

You can use `M-x check-parens` to find any unbalanced parentheses and unbalanced string quotes in the buffer.

### 23.4.1 Expressions with Balanced Parentheses

Each programming language mode has its own definition of a *balanced expression*. Balanced expressions typically include individual symbols, numbers, and string constants, as well as pieces of code enclosed in a matching pair of delimiters. The following commands deal with balanced expressions (in Emacs, such expressions are referred to internally as *sexps*[1]).

`C-M-f` Move forward over a balanced expression (`forward-sexp`).

`C-M-b` Move backward over a balanced expression (`backward-sexp`).

`C-M-k` Kill balanced expression forward (`kill-sexp`).

`C-M-t` Transpose expressions (`transpose-sexps`).

`C-M-@`
`C-M-SPC` Put mark after following expression (`mark-sexp`).

To move forward over a balanced expression, use `C-M-f` (`forward-sexp`). If the first significant character after point is an opening delimiter (e.g., '`(`', '`[`' or '`{`' in C), this command moves past the matching closing delimiter. If the character begins a symbol, string, or number, the command moves over that.

The command `C-M-b` (`backward-sexp`) moves backward over a balanced expression—like `C-M-f`, but in the reverse direction. If the expression is preceded by any prefix characters (single-quote, backquote and comma, in Lisp), the command moves back over them as well.

`C-M-f` or `C-M-b` with an argument repeats that operation the specified number of times; with a negative argument means to move in the opposite direction. In most modes, these two commands move across comments as if they were whitespace. Note that their keys, `C-M-f` and `C-M-b`, are analogous to `C-f` and `C-b`, which move by characters (see Section 4.2 [Moving Point], page 59), and `M-f` and `M-b`, which move by words (see ⟨undefined⟩ [Words], page ⟨undefined⟩).

To kill a whole balanced expression, type `C-M-k` (`kill-sexp`). This kills the text that `C-M-f` would move over.

`C-M-t` (`transpose-sexps`) switches the positions of the previous balanced expression and the next one. It is analogous to the `C-t` command, which transposes characters (see Section 13.2 [Transpose], page 218). An argument to `C-M-t` serves as a repeat count, moving the previous expression over that many following ones. A negative argument moves the previous balanced expression backwards across those before it. An argument of zero, rather than doing nothing, transposes the balanced expressions ending at or after point and the mark.

To operate on balanced expressions with a command which acts on the region, type `C-M-SPC` (`mark-sexp`). This sets the mark where `C-M-f` would move to. While the mark is active, each successive call to this command extends the region by shifting the mark by one expression. Positive or negative numeric arguments move the mark forward or backward by the specified number of expressions. The alias `C-M-@` is equivalent to `C-M-SPC`. See Section 8.2 [Marking Objects], page 90, for more information about this and related commands.

[1] The word "sexp" is used to refer to an expression in Lisp.

In languages that use infix operators, such as C, it is not possible to recognize all balanced expressions because there can be multiple possibilities at a given position. For example, C mode does not treat '`foo + bar`' as a single expression, even though it *is* one C expression; instead, it recognizes '`foo`' as one expression and '`bar`' as another, with the '`+`' as punctuation between them. However, C mode recognizes '`(foo + bar)`' as a single expression, because of the parentheses.

### 23.4.2 Moving in the Parenthesis Structure

The following commands move over groupings delimited by parentheses (or whatever else serves as delimiters in the language you are working with). They ignore strings and comments, including any parentheses within them, and also ignore parentheses that are "quoted" with an escape character. These commands are mainly intended for editing programs, but can be useful for editing any text containing parentheses. They are referred to internally as "list" commands because in Lisp these groupings are lists.

These commands assume that the starting point is not inside a string or a comment. If you invoke them from inside a string or comment, the results are unreliable.

`C-M-n` Move forward over a parenthetical group (`forward-list`).

`C-M-p` Move backward over a parenthetical group (`backward-list`).

`C-M-u` Move up in parenthesis structure (`backward-up-list`).

`C-M-d` Move down in parenthesis structure (`down-list`).

The "list" commands `C-M-n` (`forward-list`) and `C-M-p` (`backward-list`) move forward or backward over one (or *n*) parenthetical groupings.

`C-M-n` and `C-M-p` try to stay at the same level in the parenthesis structure. To move *up* one (or *n*) levels, use `C-M-u` (`backward-up-list`). `C-M-u` moves backward up past one unmatched opening delimiter. A positive argument serves as a repeat count; a negative argument reverses the direction of motion, so that the command moves forward and up one or more levels.

To move *down* in the parenthesis structure, use `C-M-d` (`down-list`). In Lisp mode, where '`(`' is the only opening delimiter, this is nearly the same as searching for a '`(`'. An argument specifies the number of levels to go down.

### 23.4.3 Matching Parentheses

Emacs has a number of *parenthesis matching* features, which make it easy to see how and whether parentheses (or other delimiters) match up.

Whenever you type a self-inserting character that is a closing delimiter, the cursor moves momentarily to the location of the matching opening delimiter, provided that is on the screen. If it is not on the screen, Emacs displays some of the text near it in the echo area. Either way, you can tell which grouping you are closing off. If the opening delimiter and closing delimiter are mismatched—such as in '`[x)`'—a warning message is displayed in the echo area.

Three variables control the display of matching parentheses:

- `blink-matching-paren` turns the feature on or off: `nil` disables it, but the default is `t` to enable it.

- `blink-matching-delay` says how many seconds to leave the cursor on the matching opening delimiter, before bringing it back to the real location of point. This may be an integer or floating-point number; the default is 1.
- `blink-matching-paren-distance` specifies how many characters back to search to find the matching opening delimiter. If the match is not found in that distance, Emacs stops scanning and nothing is displayed. The default is 102400.

Show Paren mode, a global minor mode, provides a more powerful kind of automatic matching. Whenever point is before an opening delimiter or after a closing delimiter, both that delimiter and its opposite delimiter are highlighted. To toggle Show Paren mode, type `M-x show-paren-mode`.

Electric Pair mode, a global minor mode, provides a way to easily insert matching delimiters. Whenever you insert an opening delimiter, the matching closing delimiter is automatically inserted as well, leaving point between the two. To toggle Electric Pair mode, type `M-x electric-pair-mode`.

## 23.5 Manipulating Comments

Because comments are such an important part of programming, Emacs provides special commands for editing and inserting comments. It can also do spell checking on comments with Flyspell Prog mode (see Section 13.4 [Spelling], page 219).

Some major modes have special rules for indenting different kinds of comments. For example, in Lisp code, comments starting with two semicolons are indented as if they were lines of code, while those starting with three semicolons are supposed to be aligned to the left margin and are often used for sectioning purposes. Emacs understand these conventions; for instance, typing TAB on a comment line will indent the comment to the appropriate position.

```
;; This function is just an example.
;;; Here either two or three semicolons are appropriate.
(defun foo (x)
;;;  And now, the first part of the function:
  ;; The following line adds one.
  (1+ x))           ; This line adds one.
```

### 23.5.1 Comment Commands

The following commands operate on comments:

`M-;` Insert or realign comment on current line; if the region is active, comment or uncomment the region instead (`comment-dwim`).

`C-u M-;` Kill comment on current line (`comment-kill`).

`C-x ;` Set comment column (`comment-set-column`).

`C-M-j`
`M-j` Like RET followed by inserting and aligning a comment (`comment-indent-new-line`). See Section 23.5.2 [Multi-Line Comments], page 525.

`M-x comment-region`
`C-c C-c` (in C-like modes)
Add comment delimiters to all the lines in the region.

The command to create or align a comment is `M-;` (`comment-dwim`). The word "dwim" is an acronym for "Do What I Mean"; it indicates that this command can be used for many different jobs relating to comments, depending on the situation where you use it.

When a region is active (see Chapter 8 [Mark], page 89), `M-;` either adds comment delimiters to the region, or removes them. If every line in the region is already a comment, it "uncomments" each of those lines by removing their comment delimiters. Otherwise, it adds comment delimiters to enclose the text in the region.

If you supply a prefix argument to `M-;` when a region is active, that specifies the number of comment delimiters to add or delete. A positive argument *n* adds *n* delimiters, while a negative argument *-n* removes *n* delimiters.

If the region is not active, and there is no existing comment on the current line, `M-;` adds a new comment to the current line. If the line is blank (i.e., empty or containing only whitespace characters), the comment is indented to the same position where TAB would indent to (see Section 23.3.1 [Basic Indent], page 517). If the line is non-blank, the comment is placed after the last non-whitespace character on the line; normally, Emacs tries putting it at the column specified by the variable `comment-column` (see Section 23.5.3 [Options for Comments], page 525), but if the line already extends past that column, it puts the comment at some suitable position, usually separated from the non-comment text by at least one space. In each case, Emacs places point after the comment's starting delimiter, so that you can start typing the comment text right away.

You can also use `M-;` to align an existing comment. If a line already contains the comment-start string, `M-;` realigns it to the conventional alignment and moves point after the comment's starting delimiter. As an exception, comments starting in column 0 are not moved. Even when an existing comment is properly aligned, `M-;` is still useful for moving directly to the start of the comment text.

`C-u M-;` (`comment-dwim` with a prefix argument) kills any comment on the current line, along with the whitespace before it. Since the comment is saved to the kill ring, you can reinsert it on another line by moving to the end of that line, doing `C-y`, and then `M-;` to realign the comment. You can achieve the same effect as `C-u M-;` by typing `M-x comment-kill` (`comment-dwim` actually calls `comment-kill` as a subroutine when it is given a prefix argument).

The command `M-x comment-region` is equivalent to calling `M-;` on an active region, except that it always acts on the region, even if the mark is inactive. In C mode and related modes, this command is bound to `C-c C-c`. The command `M-x uncomment-region` uncomments each line in the region; a numeric prefix argument specifies the number of comment delimiters to remove (negative arguments specify the number of comment to delimiters to add).

For C-like modes, you can configure the exact effect of `M-;` by setting the variables `c-indent-comment-alist` and `c-indent-comments-syntactically-p`. For example, on a line ending in a closing brace, `M-;` puts the comment one space after the brace rather than at `comment-column`. For full details see Section "Comment Commands" in *The CC Mode Manual*.

### 23.5.2 Multiple Lines of Comments

If you are typing a comment and wish to continue it to another line, type `M-j` or `C-M-j` (`comment-indent-new-line`). This breaks the current line, and inserts the necessary comment delimiters and indentation to continue the comment.

For languages with closing comment delimiters (e.g., '`*/`' in C), the exact behavior of `M-j` depends on the value of the variable `comment-multi-line`. If the value is `nil`, the command closes the comment on the old line and starts a new comment on the new line. Otherwise, it opens a new line within the current comment delimiters.

When Auto Fill mode is on, going past the fill column while typing a comment also continues the comment, in the same way as an explicit invocation of `M-j`.

To turn existing lines into comment lines, use `M-;` with the region active, or use `M-x comment-region` as described in the preceding section.

You can configure C Mode such that when you type a '`/`' at the start of a line in a multi-line block comment, this closes the comment. Enable the `comment-close-slash` clean-up for this. See Section "Clean-ups" in *The CC Mode Manual*.

### 23.5.3 Options Controlling Comments

As mentioned in Section 23.5.1 [Comment Commands], page 523, when the `M-j` command adds a comment to a line, it tries to place the comment at the column specified by the buffer-local variable `comment-column`. You can set either the local value or the default value of this buffer-local variable in the usual way (see Section 33.2.3 [Locals], page 697). Alternatively, you can type `C-x ;` (`comment-set-column`) to set the value of `comment-column` in the current buffer to the column where point is currently located. `C-u C-x ;` sets the comment column to match the last comment before point in the buffer, and then does a `M-;` to align the current line's comment under the previous one.

The comment commands recognize comments based on the regular expression that is the value of the variable `comment-start-skip`. Make sure this regexp does not match the null string. It may match more than the comment starting delimiter in the strictest sense of the word; for example, in C mode the value of the variable is `"\\(//+\\|/\\*+\\)\\s *"`, which matches extra stars and spaces after the '`/*`' itself, and accepts C++ style comments also. (Note that '`\\`' is needed in Lisp syntax to include a '`\`' in the string, which is needed to deny the first star its special meaning in regexp syntax. See Section 12.7 [Regexp Backslash], page 208.)

When a comment command makes a new comment, it inserts the value of `comment-start` as an opening comment delimiter. It also inserts the value of `comment-end` after point, as a closing comment delimiter. For example, in Lisp mode, `comment-start` is '`";"`' and `comment-end` is `""` (the empty string). In C mode, `comment-start` is `"/* "` and `comment-end` is `" */"`.

The variable `comment-padding` specifies a string that the commenting commands should insert between the comment delimiter(s) and the comment text. The default, '`" "`', specifies a single space. Alternatively, the value can be a number, which specifies that number of spaces, or `nil`, which means no spaces at all.

The variable `comment-multi-line` controls how `M-j` and Auto Fill mode continue comments over multiple lines. See Section 23.5.2 [Multi-Line Comments], page 525.

The variable `comment-indent-function` should contain a function that will be called to compute the alignment for a newly inserted comment or for aligning an existing comment. It is set differently by various major modes. The function is called with no arguments, but with point at the beginning of the comment, or at the end of a line if a new comment is to be inserted. It should return the column in which the comment ought to start. For example, in Lisp mode, the indent hook function bases its decision on how many semicolons begin an existing comment, and on the code in the preceding lines.

## 23.6 Documentation Lookup

Emacs provides several features you can use to look up the documentation of functions, variables and commands that you plan to use in your program.

### 23.6.1 Info Documentation Lookup

For major modes that apply to languages which have documentation in Info, you can use `C-h S` (`info-lookup-symbol`) to view the Info documentation for a symbol used in the program. You specify the symbol with the minibuffer; the default is the symbol appearing in the buffer at point. For example, in C mode this looks for the symbol in the C Library Manual. The command only works if the appropriate manual's Info files are installed.

The major mode determines where to look for documentation for the symbol—which Info files to look in, and which indices to search. You can also use `M-x info-lookup-file` to look for documentation for a file name.

If you use `C-h S` in a major mode that does not support it, it asks you to specify the "symbol help mode". You should enter a command such as `c-mode` that would select a major mode which `C-h S` does support.

### 23.6.2 Man Page Lookup

On Unix, the main form of on-line documentation was the *manual page* or *man page*. In the GNU operating system, we aim to replace man pages with better-organized manuals that you can browse with Info (see ⟨undefined⟩ [Misc Help], page ⟨undefined⟩). This process is not finished, so it is still useful to read manual pages.

You can read the man page for an operating system command, library function, or system call, with the `M-x man` command. This prompts for a topic, with completion (see Section 5.4 [Completion], page 70), and runs the `man` program to format the corresponding man page. If the system permits, it runs `man` asynchronously, so that you can keep on editing while the page is being formatted. The result goes in a buffer named '`*Man topic*`'. These buffers use a special major mode, Man mode, that facilitates scrolling and jumping to other manual pages. For details, type `C-h m` while in a Man mode buffer.

Each man page belongs to one of ten or more *sections*, each named by a digit or by a digit and a letter. Sometimes there are man pages with the same name in different sections. To read a man page from a specific section, type '`topic(section)`' or '`section topic`' when `M-x manual-entry` prompts for the topic. For example, the man page for the C library function `chmod` is in section 2, but there is a shell command of the same name, whose man page is in section 1; to view the former, type `M-x manual-entry RET chmod(2) RET`.

If you do not specify a section, `M-x man` normally displays only the first man page found. On some systems, the `man` program accepts a '`-a`' command-line option, which tells it to

display all the man pages for the specified topic. To make use of this, change the value of the variable `Man-switches` to '`"-a"`'. Then, in the Man mode buffer, you can type `M-n` and `M-p` to switch between man pages in different sections. The mode line shows how many manual pages are available.

An alternative way of reading manual pages is the `M-x woman` command. Unlike `M-x man`, it does not run any external programs to format and display the man pages; the formatting is done by Emacs, so it works on systems such as MS-Windows where the `man` program may be unavailable. It prompts for a man page, and displays it in a buffer named '`*WoMan` *section topic*'.

`M-x woman` computes the completion list for manpages the first time you invoke the command. With a numeric argument, it recomputes this list; this is useful if you add or delete manual pages.

If you type a name of a manual page and `M-x woman` finds that several manual pages by the same name exist in different sections, it pops up a window with possible candidates asking you to choose one of them.

For more information about setting up and using `M-x woman`, see the WoMan Info manual, which is distributed with Emacs.

### 23.6.3 Emacs Lisp Documentation Lookup

When editing Emacs Lisp code, you can use the commands `C-h f` (`describe-function`) and `C-h v` (`describe-variable`) to view the built-in documentation for the Lisp functions and variables that you want to use. See ⟨undefined⟩ [Name Help], page ⟨undefined⟩.

Eldoc is a buffer-local minor mode that helps with looking up Lisp documention. When it is enabled, the echo area displays some useful information whenever there is a Lisp function or variable at point; for a function, it shows the argument list, and for a variable it shows the first line of the variable's documentation string. To toggle Eldoc mode, type `M-x eldoc-mode`. Eldoc mode can be used with the Emacs Lisp and Lisp Interaction major modes.

## 23.7 Hideshow minor mode

Hideshow mode is a buffer-local minor mode that allows you to selectively display portions of a program, which are referred to as *blocks*. Type `M-x hs-minor-mode` to toggle this minor mode (see Section 20.3 [Minor Modes], page 413).

When you use Hideshow mode to hide a block, the block disappears from the screen, to be replaced by an ellipsis (three periods in a row). Just what constitutes a block depends on the major mode. In C mode and related modes, blocks are delimited by braces, while in Lisp mode they are delimited by parentheses. Multi-line comments also count as blocks.

Hideshow mode provides the following commands:

`C-c @ C-h` Hide the current block (`hs-hide-block`).

`C-c @ C-s` Show the current block (`hs-show-block`).

`C-c @ C-c` Either hide or show the current block (`hs-toggle-hiding`).

`S-Mouse-2`
: Toggle hiding for the block you click on (`hs-mouse-toggle-hiding`).

`C-c @ C-M-h`
: Hide all top-level blocks (`hs-hide-all`).

`C-c @ C-M-s`
: Show all blocks in the buffer (`hs-show-all`).

`C-c @ C-l` Hide all blocks *n* levels below this block (`hs-hide-level`).

These variables can be used to customize Hideshow mode:

`hs-hide-comments-when-hiding-all`
: If non-`nil`, `C-c @ C-M-h` (`hs-hide-all`) hides comments too.

`hs-isearch-open`
: This variable specifies the conditions under which incremental search should unhide a hidden block when matching text occurs within the block. Its value should be either `code` (unhide only code blocks), `comment` (unhide only comments), `t` (unhide both code blocks and comments), or `nil` (unhide neither code blocks nor comments). The default value is `code`.

## 23.8 Completion for Symbol Names

Completion is normally done in the minibuffer (see Section 5.4 [Completion], page 70), but you can also complete symbol names in ordinary Emacs buffers.

In programming language modes, type `C-M-i` or `M-TAB` to complete the partial symbol before point. On graphical displays, the `M-TAB` key is usually reserved by the window manager for switching graphical windows, so you should type `C-M-i` or `ESC TAB` instead.

In most programming language modes, `C-M-i` (or `M-TAB`) invokes the command `completion-at-point`, which generates its completion list in a flexible way. If Semantic mode is enabled, it tries to use the Semantic parser data for completion (see Section 23.10 [Semantic], page 529). If Semantic mode is not enabled or fails at performing completion, it tries to complete using the selected tags table (see Section 25.3 [Tags], page 571). If in Emacs Lisp mode, it performs completion using the function, variable, or property names defined in the current Emacs session.

In all other respects, in-buffer symbol completion behaves like minibuffer completion. For instance, if Emacs cannot complete to a unique symbol, it displays a list of completion alternatives in another window. See Section 5.4 [Completion], page 70.

In Text mode and related modes, `M-TAB` completes words based on the spell-checker's dictionary. See Section 13.4 [Spelling], page 219.

## 23.9 Glasses minor mode

Glasses mode is a buffer-local minor mode that makes it easier to read mixed-case (or "CamelCase") symbols like '`unReadableSymbol`', by altering how they are displayed. By default, it displays extra underscores between each lower-case letter and the following capital letter. This does not alter the buffer text, only how it is displayed.

To toggle Glasses mode, type `M-x glasses-mode` (see Section 20.3 [Minor Modes], page 413). When Glasses mode is enabled, the minor mode indicator '`o^o`' appears in the mode line. For more information about Glasses mode, type `C-h P glasses RET`.

## 23.10 Semantic

Semantic is a package that provides language-aware editing commands based on `source code parsers`. This section provides a brief description of Semantic; for full details, see the Semantic Info manual, which is distributed with Emacs.

Most of the "language aware" features in Emacs, such as Font Lock mode (see ⟨undefined⟩ [Font Lock], page ⟨undefined⟩), rely on "rules of thumb"[2] that usually give good results but are never completely exact. In contrast, the parsers used by Semantic have an exact understanding of programming language syntax. This allows Semantic to provide search, navigation, and completion commands that are powerful and precise.

To begin using Semantic, type `M-x semantic-mode` or click on the menu item named '`Source Code Parsers (Semantic)`' in the '`Tools`' menu. This enables Semantic mode, a global minor mode.

When Semantic mode is enabled, Emacs automatically attempts to parses each file you visit. Currently, Semantic understands C, C++, Scheme, Javascript, Java, HTML, and Make. Within each parsed buffer, the following commands are available:

`C-c , j` Prompt for the name of a function defined in the current file, and move point there (`semantic-complete-jump-local`).

`C-c , J` Prompt for the name of a function defined in any file Emacs has parsed, and move point there (`semantic-complete-jump`).

`C-c , SPC` Display a list of possible completions for the symbol at point (`semantic-complete-analyze-inline`). This also activates a set of special key bindings for choosing a completion: RET accepts the current completion, `M-n` and `M-p` cycle through possible completions, TAB completes as far as possible and then cycles, and `C-g` or any other key aborts completion.

`C-c , l` Display a list of the possible completions of the symbol at point, in another window (`semantic-analyze-possible-completions`).

In addition to the above commands, the Semantic package provides a variety of other ways to make use of parser information. For instance, you can use it to display a list of completions when Emacs is idle.

## 23.11 Other Features Useful for Editing Programs

Some Emacs commands that aren't designed specifically for editing programs are useful for that nonetheless.

The Emacs commands that operate on words, sentences and paragraphs are useful for editing code. Most symbols names contain words (see ⟨undefined⟩ [Words], page ⟨undefined⟩), while sentences can be found in strings and comments (see ⟨undefined⟩ [Sentences], page ⟨undefined⟩). As for paragraphs, they are defined in most programming language modes to begin and end at blank lines (see ⟨undefined⟩ [Paragraphs], page ⟨undefined⟩). Therefore, judicious use of blank lines to make the program clearer will also provide useful chunks of text for the paragraph commands to work on. Auto Fill mode, if enabled in a programming language major mode, indents the new lines which it creates.

[2] Regular expressions and syntax tables.

Electric Layout mode (`M-x electric-layout-mode`) is a global minor mode that automatically inserts newlines when you type certain characters; for example, '`{`', '`}`' and '`;`' in Javascript mode.

Apart from Hideshow mode (see Section 23.7 [Hideshow], page 527), another way to selectively display parts of a program is to use the selective display feature (see Section 11.7 [Selective Display], page 124). Programming modes often also support Outline minor mode (see ⟨undefined⟩ [Outline Mode], page ⟨undefined⟩), which can be used with the Foldout package (see ⟨undefined⟩ [Foldout], page ⟨undefined⟩).

## 23.12 C and Related Modes

This section gives a brief description of the special features available in C, C++, Objective-C, Java, CORBA IDL, Pike and AWK modes. (These are called "C mode and related modes".) For more details, see the CC mode Info manual, which is distributed with Emacs.

### 23.12.1 C Mode Motion Commands

This section describes commands for moving point, in C mode and related modes.

`C-M-a`
`C-M-e` Move point to the beginning or end of the current function or top-level definition. In languages with enclosing scopes (such as C++'s classes) the *current function* is the immediate one, possibly inside a scope. Otherwise it is the one defined by the least enclosing braces. (By contrast, `beginning-of-defun` and `end-of-defun` search for braces in column zero.) See Section 23.2.2 [Moving by Defuns], page 515.

`C-c C-u` Move point back to the containing preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move point forward to the end of the containing preprocessor conditional.

'`#elif`' is equivalent to '`#else`' followed by '`#if`', so the function will stop at a '`#elif`' when going backward, but not when going forward.

`C-c C-p` Move point back over a preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move forward.

`C-c C-n` Move point forward across a preprocessor conditional, leaving the mark behind. A prefix argument acts as a repeat count. With a negative argument, move backward.

`M-a` Move point to the beginning of the innermost C statement (`c-beginning-of-statement`). If point is already at the beginning of a statement, move to the beginning of the preceding statement. With prefix argument $n$, move back $n - 1$ statements.

In comments or in strings which span more than one line, this command moves by sentences instead of statements.

`M-e` Move point to the end of the innermost C statement or sentence; like `M-a` except that it moves in the other direction (`c-end-of-statement`).

### 23.12.2 Electric C Characters

In C mode and related modes, certain printing characters are *electric*—in addition to inserting themselves, they also reindent the current line, and optionally also insert newlines. The "electric" characters are `{`, `}`, `:`, `#`, `;`, `,`, `<`, `>`, `/`, `*`, `(`, and `)`.

You might find electric indentation inconvenient if you are editing chaotically indented code. If you are new to CC Mode, you might find it disconcerting. You can toggle electric action with the command `C-c C-l`; when it is enabled, '`/l`' appears in the mode line after the mode name:

`C-c C-l` Toggle electric action (`c-toggle-electric-state`). With a positive prefix argument, this command enables electric action, with a negative one it disables it.

Electric characters insert newlines only when, in addition to the electric state, the *auto-newline* feature is enabled (indicated by '`/la`' in the mode line after the mode name). You can turn this feature on or off with the command `C-c C-a`:

`C-c C-a` Toggle the auto-newline feature (`c-toggle-auto-newline`). With a prefix argument, this command turns the auto-newline feature on if the argument is positive, and off if it is negative.

Usually the CC Mode style configures the exact circumstances in which Emacs inserts auto-newlines. You can also configure this directly. See Section "Custom Auto-newlines" in *The CC Mode Manual*.

### 23.12.3 Hungry Delete Feature in C

If you want to delete an entire block of whitespace at point, you can use *hungry deletion*. This deletes all the contiguous whitespace either before point or after point in a single operation. *Whitespace* here includes tabs and newlines, but not comments or preprocessor commands.

`C-c C-DEL`
`C-c DEL` Delete the entire block of whitespace preceding point (`c-hungry-delete-backwards`).

`C-c C-d`
`C-c C-DELETE`
`C-c DELETE`
Delete the entire block of whitespace after point (`c-hungry-delete-forward`).

As an alternative to the above commands, you can enable *hungry delete mode*. When this feature is enabled (indicated by '`/h`' in the mode line after the mode name), a single DEL deletes all preceding whitespace, not just one space, and a single `C-c C-d` (but *not* plain DELETE) deletes all following whitespace.

`M-x c-toggle-hungry-state`
Toggle the hungry-delete feature (`c-toggle-hungry-state`). With a prefix argument, this command turns the hungry-delete feature on if the argument is positive, and off if it is negative.

The variable `c-hungry-delete-key` controls whether the hungry-delete feature is enabled.

### 23.12.4 Other Commands for C Mode

`C-c C-w`
`M-x subword-mode`
: Enable (or disable) *subword mode*. In subword mode, Emacs's word commands recognize upper case letters in '`StudlyCapsIdentifiers`' as word boundaries. This is indicated by the flag '`/w`' on the mode line after the mode name (e.g., '`C/law`'). You can even use `M-x subword-mode` in non-CC Mode buffers.

  In the GNU project, we recommend using underscores to separate words within an identifier in C or C++, rather than using case distinctions.

`M-x c-context-line-break`
: This command inserts a line break and indents the new line in a manner appropriate to the context. In normal code, it does the work of `C-j` (`newline-and-indent`), in a C preprocessor line it additionally inserts a '\' at the line break, and within comments it's like `M-j` (`c-indent-new-comment-line`).

  `c-context-line-break` isn't bound to a key by default, but it needs a binding to be useful. The following code will bind it to `C-j`. We use `c-initialization-hook` here to make sure the keymap is loaded before we try to change it.

```
(defun my-bind-clb ()
  (define-key c-mode-base-map "\C-j"
              'c-context-line-break))
(add-hook 'c-initialization-hook 'my-bind-clb)
```

`C-M-h`
: Put mark at the end of a function definition, and put point at the beginning (`c-mark-function`).

`M-q`
: Fill a paragraph, handling C and C++ comments (`c-fill-paragraph`). If any part of the current line is a comment or within a comment, this command fills the comment or the paragraph of it that point is in, preserving the comment indentation and comment delimiters.

`C-c C-e`
: Run the C preprocessor on the text in the region, and show the result, which includes the expansion of all the macro calls (`c-macro-expand`). The buffer text before the region is also included in preprocessing, for the sake of macros defined there, but the output from this part isn't shown.

  When you are debugging C code that uses macros, sometimes it is hard to figure out precisely how the macros expand. With this command, you don't have to figure it out; you can see the expansions.

`C-c C-\`
: Insert or align '\' characters at the ends of the lines of the region (`c-backslash-region`). This is useful after writing or editing a C macro definition.

  If a line already ends in '\', this command adjusts the amount of whitespace before it. Otherwise, it inserts a new '\'. However, the last line in the region is treated specially; no '\' is inserted on that line, and any '\' there is deleted.

`M-x cpp-highlight-buffer`
: Highlight parts of the text according to its preprocessor conditionals. This command displays another buffer named '`*CPP Edit*`', which serves as a graphic menu for selecting how to display particular kinds of conditionals and their

contents. After changing various settings, click on '`[A]pply these settings`' (or go to that buffer and type `a`) to rehighlight the C mode buffer accordingly.

`C-c C-s` Display the syntactic information about the current source line (`c-show-syntactic-information`). This information directs how the line is indented.

`M-x cwarn-mode`
`M-x global-cwarn-mode`
CWarn minor mode highlights certain suspicious C and C++ constructions:

- Assignments inside expressions.
- Semicolon following immediately after '`if`', '`for`', and '`while`' (except after a '`do ... while`' statement);
- C++ functions with reference parameters.

You can enable the mode for one buffer with the command `M-x cwarn-mode`, or for all suitable buffers with the command `M-x global-cwarn-mode` or by customizing the variable `global-cwarn-mode`. You must also enable Font Lock mode to make it work.

`M-x hide-ifdef-mode`
Hide-ifdef minor mode hides selected code within '`#if`' and '`#ifdef`' preprocessor blocks. If you change the variable `hide-ifdef-shadow` to `t`, Hide-ifdef minor mode "shadows" preprocessor blocks by displaying them with a less prominent face, instead of hiding them entirely. See the documentation string of `hide-ifdef-mode` for more information.

`M-x ff-find-related-file`
Find a file "related" in a special way to the file visited by the current buffer. Typically this will be the header file corresponding to a C/C++ source file, or vice versa. The variable `ff-related-file-alist` specifies how to compute related file names.

## 23.13 Asm Mode

Asm mode is a major mode for editing files of assembler code. It defines these commands:

TAB `tab-to-tab-stop`.

`C-j` Insert a newline and then indent using `tab-to-tab-stop`.

`:` Insert a colon and then remove the indentation from before the label preceding colon. Then do `tab-to-tab-stop`.

`;` Insert or align a comment.

The variable `asm-comment-char` specifies which character starts comments in assembler syntax.

# 24 Compiling and Testing Programs

The previous chapter discusses the Emacs commands that are useful for making changes in programs. This chapter deals with commands that assist in the process of compiling and testing programs.

## 24.1 Running Compilations under Emacs

Emacs can run compilers for languages such as C and Fortran, feeding the compilation log into an Emacs buffer. It can also parse the error messages and show you where the errors occurred.

`M-x compile`
: Run a compiler asynchronously under Emacs, with error messages going to the '`*compilation*`' buffer.

`M-x recompile`
: Invoke a compiler with the same command as in the last invocation of `M-x compile`.

`M-x kill-compilation`
: Kill the running compilation subprocess.

To run `make` or another compilation command, type `M-x compile`. This reads a shell command line using the minibuffer, and then executes the command by running a shell as a subprocess (or *inferior process*) of Emacs. The output is inserted in a buffer named '`*compilation*`'. The current buffer's default directory is used as the working directory for the execution of the command; normally, therefore, compilation takes place in this directory.

The default compilation command is '`make -k`', which is usually correct for programs compiled using the `make` utility (the '`-k`' flag tells `make` to continue compiling as much as possible after an error). See Section "Make" in *GNU Make Manual*. If you have done `M-x compile` before, the command that you specified is automatically stored in the variable `compile-command`; this is used as the default the next time you type `M-x compile`. A file can also specify a file-local value for `compile-command` (see Section 33.2.4 [File Variables], page 698).

Starting a compilation displays the '`*compilation*`' buffer in another window but does not select it. While the compilation is running, the word '`run`' is shown in the major mode indicator for the '`*compilation*`' buffer, and the word '`Compiling`' appears in all mode lines. You do not have to keep the '`*compilation*`' buffer visible while compilation is running; it continues in any case. When the compilation ends, for whatever reason, the mode line of the '`*compilation*`' buffer changes to say '`exit`' (followed by the exit code: '`[0]`' for a normal exit), or '`signal`' (if a signal terminated the process).

If you want to watch the compilation transcript as it appears, switch to the '`*compilation*`' buffer and move point to the end of the buffer. When point is at the end, new compilation output is inserted above point, which remains at the end. Otherwise, point remains fixed while compilation output is added at the end of the buffer.

If you change the variable `compilation-scroll-output` to a non-`nil` value, the '`*compilation*`' buffer scrolls automatically to follow the output. If the value is

`first-error`, scrolling stops when the first error appears, leaving point at that error. For any other non-`nil` value, scrolling continues until there is no more output.

To rerun the last compilation with the same command, type `M-x recompile`. This reuses the compilation command from the last invocation of `M-x compile`. It also reuses the '`*compilation*`' buffer and starts the compilation in its default directory, which is the directory in which the previous compilation was started.

Starting a new compilation also kills any compilation already running in '`*compilation*`', as the buffer can only handle one compilation at any time. However, `M-x compile` asks for confirmation before actually killing a compilation that is running; to always automatically kill the compilation without asking, change the variable `compilation-always-kill` to `t`. You can also kill a compilation process with the command `M-x kill-compilation`.

To run two compilations at once, start the first one, then rename the '`*compilation*`' buffer (perhaps using `rename-uniquely`; see ⟨undefined⟩ [Misc Buffer], page ⟨undefined⟩), then switch buffers and start the other compilation. This will create a new '`*compilation*`' buffer.

You can control the environment passed to the compilation command with the variable `compilation-environment`. Its value is a list of environment variable settings; each element should be a string of the form "*`envvarname=value`*". These environment variable settings override the usual ones.

## 24.2 Compilation Mode

The '`*compilation*`' buffer uses a major mode called Compilation mode. Compilation mode turns each error message in the buffer into a hyperlink; you can move point to it and type RET, or click on it with the mouse (see ⟨undefined⟩ [Mouse References], page ⟨undefined⟩), to visit the *locus* of the error message in a separate window. The locus is the specific position in a file where that error occurred.

If you change the variable `compilation-auto-jump-to-first-error` to a non-`nil` value, Emacs automatically visits the locus of the first error message that appears in the '`*compilation*`' buffer.

Compilation mode provides the following additional commands. These commands can also be used in '`*grep*`' buffers, where the hyperlinks are search matches rather than error messages (see Section 24.4 [Grep Searching], page 537).

`M-g M-n`
`M-g n`
`C-x ‘` Visit the locus of the next error message or match (`next-error`).

`M-g M-p`
`M-g p` Visit the locus of the previous error message or match (`previous-error`).

`M-n` Move point to the next error message or match, without visiting its locus (`compilation-next-error`).

`M-p` Move point to the previous error message or match, without visiting its locus (`compilation-previous-error`).

`M-}` Move point to the next error message or match occurring in a different file (`compilation-next-file`).

`M-{` Move point to the previous error message or match occurring in a different file (`compilation-previous-file`).

`C-c C-f` Toggle Next Error Follow minor mode, which makes cursor motion in the compilation buffer produce automatic source display.

To visit errors sequentially, type `C-x ‘` (`next-error`), or equivalently `M-g M-n` or `M-g n`. This command can be invoked from any buffer, not just a Compilation mode buffer. The first time you invoke it after a compilation, it visits the locus of the first error message. Each subsequent `C-x ‘` visits the next error, in a similar fashion. If you visit a specific error with RET or a mouse click in the ‘`*compilation*`’ buffer, subsequent `C-x ‘` commands advance from there. When `C-x ‘` finds no more error messages to visit, it signals an error. `C-u C-x ‘` starts again from the beginning of the compilation buffer, and visits the first locus.

`M-g M-p` or `M-g p` (`previous-error`) iterates through errors in the opposite direction.

The `next-error` and `previous-error` commands don’t just act on the errors or matches listed in ‘`*compilation*`’ and ‘`*grep*`’ buffers; they also know how to iterate through error or match lists produced by other commands, such as `M-x occur` (see Section 12.11 [Other Repeating Search], page 215). If you are already in a buffer containing error messages or matches, those are the ones that are iterated through; otherwise, Emacs looks for a buffer containing error messages or matches amongst the windows of the selected frame, then for one that `next-error` or `previous-error` previously iterated through, and finally amongst all other buffers. If the buffer chosen for iterating through is not currently displayed in a window, it will be displayed.

By default, the `next-error` and `previous-error` commands skip less important messages. The variable `compilation-skip-threshold` controls this. The default value, 1, means to skip anything less important than a warning. A value of 2 means to skip anything less important than an error, while 0 means not to skip any messages.

When Emacs visits the locus of an error message, it momentarily highlights the relevant source line. The duration of this highlight is determined by the variable `next-error-highlight`.

If the ‘`*compilation*`’ buffer is shown in a window with a left fringe (see Section 11.13 [Fringes], page 156), the locus-visiting commands put an arrow in the fringe, pointing to the current error message. If the window has no left fringe, such as on a text terminal, these commands scroll the window so that the current message is at the top of the window. If you change the variable `compilation-context-lines` to an integer value *n*, these commands scroll the window so that the current error message is *n* lines from the top, whether or not there is a fringe; the default value, `nil`, gives the behavior described above.

To parse messages from the compiler, Compilation mode uses the variable `compilation-error-regexp-alist` which lists various error message formats and tells Emacs how to extract the locus from each. A similar variable, `grep-regexp-alist`, tells Emacs how to parse output from a `grep` command (see Section 24.4 [Grep Searching], page 537).

Compilation mode also defines the keys SPC and DEL to scroll by screenfuls; `M-n` (`compilation-next-error`) and `M-p` (`compilation-previous-error`) to move to the next

or previous error message; and `M-{` (`compilation-next-file`) and `M-}` (`compilation-previous-file`) to move to the next or previous error message for a different source file.

You can type `C-c C-f` to toggle Next Error Follow mode. In this minor mode, ordinary cursor motion in the compilation buffer automatically updates the source buffer, i.e., moving the cursor over an error message causes the locus of that error to be displayed.

The features of Compilation mode are also available in a minor mode called Compilation Minor mode. This lets you parse error messages in any buffer, not just a normal compilation output buffer. Type `M-x compilation-minor-mode` to enable the minor mode. For instance, in an Rlogin buffer (see Section 31.3.10 [Remote Host], page 664), Compilation minor mode automatically accesses remote source files by FTP (see Section 15.8 [File Names], page 251).

## 24.3 Subshells for Compilation

The `M-x compile` command uses a shell to run the compilation command, but specifies the option for a noninteractive shell. This means, in particular, that the shell should start with no prompt. If you find your usual shell prompt making an unsightly appearance in the '`*compilation*`' buffer, it means you have made a mistake in your shell's init file by setting the prompt unconditionally. (This init file may be named '`.bashrc`', '`.profile`', '`.cshrc`', '`.shrc`', etc., depending on what shell you use.) The shell init file should set the prompt only if there already is a prompt. Here's how to do it in bash:

```
if [ "${PS1+set}" = set ]
then PS1=...
fi
```

And here's how to do it in csh:

```
if ($?prompt) set prompt = ...
```

Emacs does not expect a compiler process to launch asynchronous subprocesses; if it does, and they keep running after the main compiler process has terminated, Emacs may kill them or their output may not arrive in Emacs. To avoid this problem, make the main compilation process wait for its subprocesses to finish. In a shell script, you can do this using '`$!`' and '`wait`', like this:

```
(sleep 10; echo 2nd)& pid=$!  # Record pid of subprocess
echo first message
wait $pid                     # Wait for subprocess
```

If the background process does not output to the compilation buffer, so you only need to prevent it from being killed when the main compilation process terminates, this is sufficient:

```
nohup command; sleep 1
```

## 24.4 Searching with Grep under Emacs

Just as you can run a compiler from Emacs and then visit the lines with compilation errors, you can also run `grep` and then visit the lines on which matches were found. This works by treating the matches reported by `grep` as if they were "errors". The output buffer uses Grep mode, which is a variant of Compilation mode (see Section 24.2 [Compilation Mode], page 535).

`M-x grep`
`M-x lgrep` Run `grep` asynchronously under Emacs, listing matching lines in the buffer named '`*grep*`'.

`M-x grep-find`
`M-x find-grep`
`M-x rgrep` Run `grep` via `find`, and collect output in the '`*grep*`' buffer.

`M-x zrgrep`
Run `zgrep` and collect output in the '`*grep*`' buffer.

`M-x kill-grep`
Kill the running `grep` subprocess.

To run `grep`, type `M-x grep`, then enter a command line that specifies how to run `grep`. Use the same arguments you would give `grep` when running it normally: a `grep`-style regexp (usually in single-quotes to quote the shell's special characters) followed by file names, which may use wildcards. If you specify a prefix argument for `M-x grep`, it finds the tag (see Section 25.3 [Tags], page 571) in the buffer around point, and puts that into the default `grep` command.

Your command need not simply run `grep`; you can use any shell command that produces output in the same format. For instance, you can chain `grep` commands, like this:

```
grep -nH -e foo *.el | grep bar | grep toto
```

The output from `grep` goes in the '`*grep*`' buffer. You can find the corresponding lines in the original files using `C-x '`, RET, and so forth, just like compilation errors.

Some grep programs accept a '`--color`' option to output special markers around matches for the purpose of highlighting. You can make use of this feature by setting `grep-highlight-matches` to `t`. When displaying a match in the source buffer, the exact match will be highlighted, instead of the entire source line.

The command `M-x grep-find` (also available as `M-x find-grep`) is similar to `M-x grep`, but it supplies a different initial default for the command—one that runs both `find` and `grep`, so as to search every file in a directory tree. See also the `find-grep-dired` command, in Section 27.15 [Dired and Find], page 600.

The commands `M-x lgrep` (local grep) and `M-x rgrep` (recursive grep) are more user-friendly versions of `grep` and `grep-find`, which prompt separately for the regular expression to match, the files to search, and the base directory for the search. Case sensitivity of the search is controlled by the current value of `case-fold-search`. The command `M-x zrgrep` is similar to `M-x rgrep`, but it calls `zgrep` instead of `grep` to search the contents of gzipped files.

These commands build the shell commands based on the variables `grep-template` (for `lgrep`) and `grep-find-template` (for `rgrep`). The files to search can use aliases defined in the variable `grep-files-aliases`.

Directories listed in the variable `grep-find-ignored-directories` are automatically skipped by `M-x rgrep`. The default value includes the data directories used by various version control systems.

## 24.5 Finding Syntax Errors On The Fly

Flymake mode is a minor mode that performs on-the-fly syntax checking for many programming and markup languages, including C, C++, Perl, HTML, and TEX/LATEX. It is somewhat analogous to Flyspell mode, which performs spell checking for ordinary human languages in a similar fashion (see Section 13.4 [Spelling], page 219). As you edit a file, Flymake mode runs an appropriate syntax checking tool in the background, using a temporary copy of the buffer. It then parses the error and warning messages, and highlights the erroneous lines in the buffer. The syntax checking tool used depends on the language; for example, for C/C++ files this is usually the C compiler. Flymake can also use build tools such as `make` for checking complicated projects.

To enable Flymake mode, type `M-x flymake-mode`. You can jump to the errors that it finds by using `M-x flymake-goto-next-error` and `M-x flymake-goto-prev-error`. To display any error messages associated with the current line, type `M-x flymake-display-err-menu-for-current-line`.

For more details about using Flymake, see the Flymake Info manual, which is distributed with Emacs.

## 24.6 Running Debuggers Under Emacs

The GUD (Grand Unified Debugger) library provides an Emacs interface to a wide variety of symbolic debuggers. It can run the GNU Debugger (GDB), as well as DBX, SDB, XDB, Perl's debugging mode, the Python debugger PDB, and the Java Debugger JDB.

Emacs provides a special interface to GDB, which uses extra Emacs windows to display the state of the debugged program. See Section 24.6.5 [GDB Graphical Interface], page 543.

Emacs also has a built-in debugger for Emacs Lisp programs. See Section "The Lisp Debugger" in *the Emacs Lisp Reference Manual*.

### 24.6.1 Starting GUD

There are several commands for starting a debugger subprocess, each corresponding to a particular debugger program.

`M-x gdb` Run GDB as a subprocess, and interact with it via an IDE-like Emacs interface. See Section 24.6.5 [GDB Graphical Interface], page 543, for more information about this command.

`M-x gud-gdb`
Run GDB, using a GUD interaction buffer for input and output to the GDB subprocess (see Section 24.6.2 [Debugger Operation], page 540). If such a buffer already exists, switch to it; otherwise, create the buffer and switch to it.

The other commands in this list do the same, for other debugger programs.

`M-x perldb`
Run the Perl interpreter in debug mode.

`M-x jdb` Run the Java debugger.

`M-x pdb` Run the Python debugger.

`M-x dbx` Run the DBX debugger.

`M-x xdb` Run the XDB debugger.

`M-x sdb` Run the SDB debugger.

Each of these commands reads a command line to invoke the debugger, using the minibuffer. The minibuffer's initial contents contain the standard executable name and options for the debugger, and sometimes also a guess for the name of the executable file you want to debug. Shell wildcards and variables are not allowed in this command line. Emacs assumes that the first command argument which does not start with a '`-`' is the executable file name.

Tramp provides a facility for remote debugging, whereby both the debugger and the program being debugged are on the same remote host. See Section "Running a debugger on a remote host" in *The Tramp Manual*, for details. This is separate from GDB's remote debugging feature, where the program and the debugger run on different machines (see Section "Debugging Remote Programs" in *The GNU debugger*).

### 24.6.2 Debugger Operation

The *GUD interaction buffer* is an Emacs buffer which is used to send text commands to a debugger subprocess, and record its output. This is the basic interface for interacting with a debugger, used by `M-x gud-gdb` and other commands listed in the preceding section. The `M-x gdb` command extends this interface with additional specialized buffers for controlling breakpoints, stack frames, and other aspects of the debugger state (see Section 24.6.5 [GDB Graphical Interface], page 543).

The GUD interaction buffer uses a variant of Shell mode, so the Emacs commands defined by Shell mode are available (see Section 31.3.3 [Shell Mode], page 657). Completion is available for most debugger commands (see Section 5.4 [Completion], page 70), and you can use the usual Shell mode history commands to repeat them. See the next section for special commands that can be used in the GUD interaction buffer.

As you debug a program, Emacs displays the relevant source files by visiting them in Emacs buffers, with an arrow in the left fringe indicating the current execution line. (On a text terminal, the arrow appears as '`=>`', overlaid on the first two text columns.) Moving point in such a buffer does not move the arrow. You are free to edit these source files, but note that inserting or deleting lines will throw off the arrow's positioning, as Emacs has no way to figure out which edited source line corresponds to the line reported by the debugger subprocess. To update this information, you typically have to recompile and restart the program.

GUD Tooltip mode is a global minor mode that adds tooltip support to GUD. To toggle this mode, type `M-x gud-tooltip-mode`. It is disabled by default. If enabled, you can move the mouse cursor over a variable, a function, or a macro (collectively called *identifiers*) to show their values in tooltips (see ⟨undefined⟩ [Tooltips], page ⟨undefined⟩). Alternatively, mark an identifier or an expression by dragging the mouse over it, then leave the mouse in the marked area to have the value of the expression displayed in a tooltip. The GUD Tooltip mode takes effect in the GUD interaction buffer, and in all source buffers with major modes listed in the variable `gud-tooltip-modes`. If the variable `gud-tooltip-echo-area` is non-`nil`, or if you turned off the tooltip mode, values are shown in the echo area instead of a tooltip.

When using GUD Tooltip mode with `M-x gud-gdb`, displaying an expression's value in GDB can sometimes expand a macro, potentially causing side effects in the debugged

program. For that reason, using tooltips in `gud-gdb` is disabled. If you use the `M-x gdb` interface, this problem does not occur, as there is special code to avoid side-effects; furthermore, you can display macro definitions associated with an identifier when the program is not executing.

### 24.6.3 Commands of GUD

GUD provides commands for setting and clearing breakpoints, selecting stack frames, and stepping through the program.

`C-x SPC` Set a breakpoint on the source line that point is on.

`C-x SPC` (`gud-break`), when called in a source buffer, sets a debugger breakpoint on the current source line. This command is available only after starting GUD. If you call it in a buffer that is not associated with any debugger subprocess, it signals a error.

The following commands are available both in the GUD interaction buffer and globally, but with different key bindings. The keys starting with `C-c` are available only in the GUD interaction buffer, while those starting with `C-x C-a` are available globally. Some of these commands are also available via the tool bar; some are not supported by certain debuggers.

`C-c C-l`
`C-x C-a C-l`
: Display, in another window, the last source line referred to in the GUD interaction buffer (`gud-refresh`).

`C-c C-s`
`C-x C-a C-s`
: Execute the next single line of code (`gud-step`). If the line contains a function call, execution stops after entering the called function.

`C-c C-n`
`C-x C-a C-n`
: Execute the next single line of code, stepping across function calls without stopping inside the functions (`gud-next`).

`C-c C-i`
`C-x C-a C-i`
: Execute a single machine instruction (`gud-stepi`).

`C-c C-p`
`C-x C-a C-p`
: Evaluate the expression at point (`gud-print`). If Emacs does not print the exact expression that you want, mark it as a region first.

`C-c C-r`
`C-x C-a C-r`
: Continue execution without specifying any stopping point. The program will run until it hits a breakpoint, terminates, or gets a signal that the debugger is checking for (`gud-cont`).

`C-c C-d`
`C-x C-a C-d`
: Delete the breakpoint(s) on the current source line, if any (`gud-remove`). If you use this command in the GUD interaction buffer, it applies to the line where the program last stopped.

`C-c C-t`
`C-x C-a C-t`
: Set a temporary breakpoint on the current source line, if any (`gud-tbreak`). If you use this command in the GUD interaction buffer, it applies to the line where the program last stopped.

`C-c <`
`C-x C-a <` Select the next enclosing stack frame (`gud-up`). This is equivalent to the GDB command '`up`'.

`C-c >`
`C-x C-a >` Select the next inner stack frame (`gud-down`). This is equivalent to the GDB command '`down`'.

`C-c C-u`
`C-x C-a C-u`
: Continue execution to the current line (`gud-until`). The program will run until it hits a breakpoint, terminates, gets a signal that the debugger is checking for, or reaches the line on which the cursor currently sits.

`C-c C-f`
`C-x C-a C-f`
: Run the program until the selected stack frame returns or stops for some other reason (`gud-finish`).

If you are using GDB, these additional key bindings are available:

`C-x C-a C-j`
: Only useful in a source buffer, `gud-jump` transfers the program's execution point to the current line. In other words, the next line that the program executes will be the one where you gave the command. If the new execution line is in a different function from the previously one, GDB prompts for confirmation since the results may be bizarre. See the GDB manual entry regarding `jump` for details.

TAB With GDB, complete a symbol name (`gud-gdb-complete-command`). This key is available only in the GUD interaction buffer.

These commands interpret a numeric argument as a repeat count, when that makes sense.

Because TAB serves as a completion command, you can't use it to enter a tab as input to the program you are debugging with GDB. Instead, type `C-q TAB` to enter a tab.

### 24.6.4 GUD Customization

On startup, GUD runs one of the following hooks: `gdb-mode-hook`, if you are using GDB; `dbx-mode-hook`, if you are using DBX; `sdb-mode-hook`, if you are using SDB; `xdb-mode-hook`, if you are using XDB; `perldb-mode-hook`, for Perl debugging mode; `pdb-mode-hook`, for PDB; `jdb-mode-hook`, for JDB. See Section 33.2.2 [Hooks], page 696.

The `gud-def` Lisp macro (see Section "Defining Macros" in *the Emacs Lisp Reference Manual*) provides a convenient way to define an Emacs command that sends a particular command string to the debugger, and set up a key binding for in the GUD interaction buffer:

```
(gud-def function cmdstring binding docstring)
```

This defines a command named *function* which sends *cmdstring* to the debugger process, and gives it the documentation string *docstring*. You can then use the command *function* in any buffer. If *binding* is non-`nil`, `gud-def` also binds the command to `C-c` *binding* in the GUD buffer's mode and to `C-x C-a` *binding* generally.

The command string *cmdstring* may contain certain '`%`'-sequences that stand for data to be filled in at the time *function* is called:

'`%f`' The name of the current source file. If the current buffer is the GUD buffer, then the "current source file" is the file that the program stopped in.

'`%l`' The number of the current source line. If the current buffer is the GUD buffer, then the "current source line" is the line that the program stopped in.

'`%e`' In transient-mark-mode the text in the region, if it is active. Otherwise the text of the C lvalue or function-call expression at or adjacent to point.

'`%a`' The text of the hexadecimal address at or adjacent to point.

'`%p`' The numeric argument of the called function, as a decimal number. If the command is used without a numeric argument, '`%p`' stands for the empty string.

If you don't use '`%p`' in the command string, the command you define ignores any numeric argument.

'`%d`' The name of the directory of the current source file.

'`%c`' Fully qualified class name derived from the expression surrounding point (jdb only).

### 24.6.5 GDB Graphical Interface

The command `M-x gdb` starts GDB in an IDE-like interface, with specialized buffers for controlling breakpoints, stack frames, and other aspects of the debugger state. It also provides additional ways to control the debugging session with the mouse, such as clicking in the fringe of a source buffer to set a breakpoint there.

To run GDB using just the GUD interaction buffer interface, without these additional features, use `M-x gud-gdb` (see Section 24.6.1 [Starting GUD], page 539). You must use this if you want to debug multiple programs within one Emacs session, as that is currently unsupported by `M-x gdb`.

Internally, `M-x gdb` informs GDB that its "screen size" is unlimited; for correct operation, you must not change GDB's screen height and width values during the debugging session.

#### 24.6.5.1 GDB User Interface Layout

If the variable `gdb-many-windows` is `nil` (the default), `M-x gdb` normally displays only the GUD interaction buffer. However, if the variable `gdb-show-main` is also non-`nil`, it starts with two windows: one displaying the GUD interaction buffer, and the other showing the source for the `main` function of the program you are debugging.

If `gdb-many-windows` is non-`nil`, then `M-x gdb` displays the following frame layout:

```
+--------------------------------+--------------------------------+
|   GUD interaction buffer       |   Locals/Registers buffer      |
|--------------------------------+--------------------------------+
|   Primary Source buffer        |   I/O buffer for debugged pgm  |
|--------------------------------+--------------------------------+
|   Stack buffer                 |   Breakpoints/Threads buffer   |
+--------------------------------+--------------------------------+
```

If you ever change the window layout, you can restore the "many windows" layout by typing `M-x gdb-restore-windows`. To toggle between the many windows layout and a simple layout with just the GUD interaction buffer and a source file, type `M-x gdb-many-windows`.

You may also specify additional GDB-related buffers to display, either in the same frame or a different one. Select the buffers you want by typing `M-x gdb-display-`*`buffertype`*`-buffer` or `M-x gdb-frame-`*`buffertype`*`-buffer`, where *buffertype* is the relevant buffer type, such as '`breakpoints`'. You can do the same with the menu bar, with the '`GDB-Windows`' and '`GDB-Frames`' sub-menus of the '`GUD`' menu.

When you finish debugging, kill the GUD interaction buffer with `C-x k`, which will also kill all the buffers associated with the session. However you need not do this if, after editing and re-compiling your source code within Emacs, you wish to continue debugging. When you restart execution, GDB automatically finds the new executable. Keeping the GUD interaction buffer has the advantage of keeping the shell history as well as GDB's breakpoints. You do need to check that the breakpoints in recently edited source files are still in the right places.

#### 24.6.5.2 Source Buffers

`Mouse-1` (in fringe)
: Set or clear a breakpoint on that line.

`C-Mouse-1` (in fringe)
: Enable or disable a breakpoint on that line.

`Mouse-3` (in fringe)
: Continue execution to that line.

`C-Mouse-3` (in fringe)
: Jump to that line.

On a graphical display, you can click `Mouse-1` in the fringe of a source buffer, to set a breakpoint on that line (see Section 11.13 [Fringes], page 156). A red dot appears in the fringe, where you clicked. If a breakpoint already exists there, the click removes it. A `C-Mouse-1` click enables or disables an existing breakpoint; a breakpoint that is disabled, but not unset, is indicated by a gray dot.

On a text terminal, or when fringes are disabled, enabled breakpoints are indicated with a '`B`' character in the left margin of the window. Disabled breakpoints are indicated with '`b`'. (The margin is only displayed if a breakpoint is present.)

A solid arrow in the left fringe of a source buffer indicates the line of the innermost frame where the debugged program has stopped. A hollow arrow indicates the current execution line of a higher-level frame. If you drag the arrow in the fringe with `Mouse-1`, that causes execution to advance to the line where you release the button. Alternatively, you can click `Mouse-3` in the fringe to advance to that line. You can click `C-Mouse-3` in the fringe to jump to that line without executing the intermediate lines. This command allows you to go backwards, which can be useful for running through code that has already executed, in order to examine its execution in more detail.

#### 24.6.5.3 Breakpoints Buffer

The GDB Breakpoints buffer shows the breakpoints, watchpoints and catchpoints in the debugger session. See Section "Breakpoints" in *The GNU debugger*. It provides the following commands, which mostly apply to the *current breakpoint* (the breakpoint which point is on):

SPC — Enable/disable current breakpoint (`gdb-toggle-breakpoint`). On a graphical display, this changes the color of the dot in the fringe of the source buffer at that line. The dot is red when the breakpoint is enabled, and gray when it is disabled.

`D` — Delete the current breakpoint (`gdb-delete-breakpoint`).

RET — Visit the source line for the current breakpoint (`gdb-goto-breakpoint`).

`Mouse-2` — Visit the source line for the breakpoint you click on.

When `gdb-many-windows` is non-`nil`, the GDB Breakpoints buffer shares its window with the GDB Threads buffer. To switch from one to the other click with `Mouse-1` on the relevant button in the header line. If `gdb-show-threads-by-default` is non-`nil`, the GDB Threads buffer is the one shown by default.

#### 24.6.5.4 Threads Buffer

The GDB Threads buffer displays a summary of the threads in the debugged program. See Section "Debugging programs with multiple threads" in *The GNU debugger*. To select a thread, move point there and type RET (`gdb-select-thread`), or click on it with `Mouse-2`. This also displays the associated source buffer, and updates the contents of the other GDB buffers.

You can customize variables under `gdb-buffers` group to select fields included in GDB Threads buffer.

`gdb-thread-buffer-verbose-names`
: Show long thread names like '`Thread 0x4e2ab70 (LWP 1983)`'.

`gdb-thread-buffer-arguments`
: Show arguments of thread top frames.

`gdb-thread-buffer-locations`
: Show file information or library names.

`gdb-thread-buffer-addresses`
: Show addresses for thread frames in threads buffer.

To view information for several threads simultaneously, use the following commands from the GDB Threads buffer.

`d`
: Display disassembly buffer for the thread at current line (`gdb-display-disassembly-for-thread`).

`f`
: Display the GDB Stack buffer for the thread at current line (`gdb-display-stack-for-thread`).

`l`
: Display the GDB Locals buffer for the thread at current line (`gdb-display-locals-for-thread`).

`r`
: Display the GDB Registers buffer for the thread at current line (`gdb-display-registers-for-thread`).

Their upper-case counterparts, `D`, `F` ,`L` and `R`, display the corresponding buffer in a new frame.

When you create a buffer showing information about some specific thread, it becomes bound to that thread and keeps showing actual information while you debug your program. The mode indicator for each GDB buffer shows the number of thread it is showing information about. The thread number is also included in the buffer name of bound buffers.

Further commands are available in the GDB Threads buffer which depend on the mode of GDB that is used for controlling execution of your program. See Section 24.6.5.8 [Multithreaded Debugging], page 548.

### 24.6.5.5 Stack Buffer

The GDB Stack buffer displays a *call stack*, with one line for each of the nested subroutine calls (*stack frames*) in the debugger session. See Section "Backtraces" in *The GNU debugger*.

On graphical displays, the selected stack frame is indicated by an arrow in the fringe. On text terminals, or when fringes are disabled, the selected stack frame is displayed in reverse contrast. To select a stack frame, move point in its line and type RET (`gdb-frames-select`), or click `Mouse-2` on it. Doing so also updates the Locals buffer (described in the next section).

### 24.6.5.6 Other GDB Buffers

Locals Buffer
: This buffer displays the values of local variables of the current frame for simple data types (see Section "Information on a frame" in *The GNU debugger*). Press RET or click `Mouse-2` on the value if you want to edit it.

  Arrays and structures display their type only. With GDB 6.4 or later, you can examine the value of the local variable at point by typing RET, or with a `Mouse-2` click. With earlier versions of GDB, use RET or `Mouse-2` on the type description ('`[struct/union]`' or '`[array]`'). See Section 24.6.5.7 [Watch Expressions], page 547.

Registers Buffer
: This buffer displays the values held by the registers (see Section "Registers" in *The GNU debugger*). Press RET or click `Mouse-2` on a register if you want to edit its value. With GDB 6.4 or later, recently changed register values display with `font-lock-warning-face`.

Assembler Buffer
: The assembler buffer displays the current frame as machine code. An arrow points to the current instruction, and you can set and remove breakpoints as in a source buffer. Breakpoint icons also appear in the fringe or margin.

Memory Buffer
: The memory buffer lets you examine sections of program memory (see Section "Examining memory" in *The GNU debugger*). Click `Mouse-1` on the appropriate part of the header line to change the starting address or number of data items that the buffer displays. Alternatively, use `S` or `N` respectively. Click `Mouse-3` on the header line to select the display format or unit size for these data items.

When `gdb-many-windows` is non-`nil`, the locals buffer shares its window with the registers buffer, just like breakpoints and threads buffers. To switch from one to the other, click with `Mouse-1` on the relevant button in the header line.

#### 24.6.5.7 Watch Expressions

If you want to see how a variable changes each time your program stops, move point into the variable name and click on the watch icon in the tool bar (`gud-watch`) or type `C-x C-a C-w`. If you specify a prefix argument, you can enter the variable name in the minibuffer.

Each watch expression is displayed in the speedbar (see ⟨undefined⟩ [Speedbar], page ⟨undefined⟩). Complex data types, such as arrays, structures and unions are represented in a tree format. Leaves and simple data types show the name of the expression and its value and, when the speedbar frame is selected, display the type as a tooltip. Higher levels show the name, type and address value for pointers and just the name and type otherwise. Root expressions also display the frame address as a tooltip to help identify the frame in which they were defined.

To expand or contract a complex data type, click `Mouse-2` or press SPC on the tag to the left of the expression. Emacs asks for confirmation before expanding the expression if its number of immediate children exceeds the value of the variable `gdb-max-children`.

To delete a complex watch expression, move point to the root expression in the speedbar and type `D` (`gdb-var-delete`).

To edit a variable with a simple data type, or a simple element of a complex data type, move point there in the speedbar and type RET (`gdb-edit-value`). Or you can click `Mouse-2` on a value to edit it. Either way, this reads the new value using the minibuffer.

If you set the variable `gdb-show-changed-values` to non-`nil` (the default value), Emacs uses `font-lock-warning-face` to highlight values that have recently changed and `shadow` face to make variables which have gone out of scope less noticeable. When a variable goes out of scope you can't edit its value.

If the variable `gdb-delete-out-of-scope` is non-`nil` (the default value), Emacs automatically deletes watch expressions which go out of scope. Sometimes, when re-entering the

same function, it may be useful to set this value to `nil` so that you don't need to recreate the watch expression.

If the variable `gdb-use-colon-colon-notation` is non-`nil`, Emacs uses the '*`function::variable`*' format. This allows the user to display watch expressions which share the same variable name. The default value is `nil`.

To automatically raise the speedbar every time the display of watch expressions updates, set `gdb-speedbar-auto-raise` to non-`nil`. This can be useful if you are debugging with a full screen Emacs frame.

#### 24.6.5.8 Multithreaded Debugging

In GDB's *all-stop mode*, whenever your program stops, all execution threads stop. Likewise, whenever you restart the program, all threads start executing. See Section "All-Stop Mode" in *The GNU debugger*. For some multi-threaded targets, GDB supports a further mode of operation, called *non-stop mode*, in which you can examine stopped program threads in the debugger while other threads continue to execute freely. See Section "Non-Stop Mode" in *The GNU debugger*. Versions of GDB prior to 7.0 do not support non-stop mode, and it does not work on all targets.

The variable `gdb-non-stop-setting` determines whether Emacs runs GDB in all-stop mode or non-stop mode. The default is `t`, which means it tries to use non-stop mode if that is available. If you change the value to `nil`, or if non-stop mode is unavailable, Emacs runs GDB in all-stop mode. The variable takes effect when Emacs begins a debugging session; if you change its value, you should restart any active debugging session.

When a thread stops in non-stop mode, Emacs usually switches to that thread. If you don't want Emacs to do this switch if another stopped thread is already selected, change the variable `gdb-switch-when-another-stopped` to `nil`.

Emacs can decide whether or not to switch to the stopped thread depending on the reason which caused the stop. Customize the variable `gdb-switch-reasons` to select the stop reasons which will cause a thread switch.

The variable `gdb-stopped-functions` allows you to execute your functions whenever some thread stops.

In non-stop mode, you can switch between different modes for GUD execution control commands.

*Non-stop/A*
: When `gdb-gud-control-all-threads` is `t` (the default value), interruption and continuation commands apply to all threads, so you can halt or continue all your threads with one command using `gud-stop-subjob` and `gud-cont`, respectively. The '`Go`' button is shown on the toolbar when at least one thread is stopped, whereas '`Stop`' button is shown when at least one thread is running.

*Non-stop/T*
: When `gdb-gud-control-all-threads` is `nil`, only the current thread is stopped/continued. '`Go`' and '`Stop`' buttons on the GUD toolbar are shown depending on the state of current thread.

You can change the current value of `gdb-gud-control-all-threads` from the tool bar or from '`GUD->GDB-MI`' menu.

Stepping commands always apply to the current thread.

In non-stop mode, you can interrupt/continue your threads without selecting them. Hitting `i` in threads buffer interrupts thread under point, `c` continues it, `s` steps through. More such commands may be added in the future.

Note that when you interrupt a thread, it stops with the '`signal received`' reason. If that reason is included in your `gdb-switch-reasons` (it is by default), Emacs will switch to that thread.

## 24.7 Executing Lisp Expressions

Emacs has major modes for several variants of Lisp. They use the same editing commands as other programming language modes (see Chapter 23 [Programs], page 514). In addition, they provide special commands for executing Lisp expressions.

Emacs Lisp mode
: The mode for editing Emacs Lisp source files. It defines `C-M-x` to evaluate the current top-level Lisp expression. See Section 24.9 [Lisp Eval], page 550.

Lisp Interaction mode
: The mode for an interactive Emacs Lisp session. It defines `C-j` to evaluate the expression before point and insert its value in the buffer. See Section 24.10 [Lisp Interaction], page 552.

Lisp mode
: The mode for editing source files of programs that run in Lisps other than Emacs Lisp. It defines `C-M-x` to evaluate the current top-level expression in an external Lisp. See Section 24.11 [External Lisp], page 552.

Inferior Lisp mode
: The mode for an interactive session with an external Lisp which is being run as a subprocess (or *inferior process*) of Emacs.

Scheme mode
: Like Lisp mode, but for Scheme programs.

Inferior Scheme mode
: Like Inferior Lisp mode, but for Scheme.

## 24.8 Libraries of Lisp Code for Emacs

Emacs Lisp code is stored in files whose names conventionally end in '`.el`'. Such files are automatically visited in Emacs Lisp mode.

Emacs Lisp code can be compiled into byte-code, which loads faster, takes up less space, and executes faster. By convention, compiled Emacs Lisp code goes in a separate file whose name ends in '`.elc`'. For example, the compiled code for '`foo.el`' goes in '`foo.elc`'. See Section "Byte Compilation" in *the Emacs Lisp Reference Manual*.

To *load* an Emacs Lisp file, type `M-x load-file`. This command reads a file name using the minibuffer, and executes the contents of that file as Emacs Lisp code. It is not necessary to visit the file first; this command reads the file directly from disk, not from an existing Emacs buffer.

If an Emacs Lisp file is installed in the Emacs Lisp *load path* (defined below), you can load it by typing `M-x load-library`, instead of using `M-x load-file`. The `M-x load-library` command prompts for a *library name* rather than a file name; it searches through each directory in the Emacs Lisp load path, trying to find a file matching that library name. If the library name is '*foo*', it tries looking for files named '*foo*`.elc`', '*foo*`.el`', and lastly just '*foo*'; the first one found is loaded. This command prefers '`.elc`' files over '`.el`' files because compiled files load and run faster. If it finds that '*lib*`.el`' is newer than '*lib*`.elc`', it issues a warning, in case someone made changes to the '`.el`' file and forgot to recompile it, but loads the '`.elc`' file anyway. (Due to this behavior, you can save unfinished edits to Emacs Lisp source files, and not recompile until your changes are ready for use.)

Emacs Lisp programs usually load Emacs Lisp files using the `load` function. This is similar to `load-library`, but is lower-level and accepts additional arguments. See Section "How Programs Do Loading" in *the Emacs Lisp Reference Manual*.

The Emacs Lisp load path is specified by the variable `load-path`. Its value should be a list of directory names (strings). These directories are searched, in the specified order, by the `M-x load-library` command, the lower-level `load` function, and other Emacs functions that find Emacs Lisp libraries. A list entry in `load-path` can also have the special value `nil`, which stands for the current default directory, but it is almost always a bad idea to use this. (If you find yourself wishing that `nil` were in the list, most likely what you really want is to use `M-x load-file`.)

The default value of `load-path` is a list of directories where the Lisp code for Emacs itself is stored. If you have libraries of your own in another directory, you can add that directory to the load path. Unlike most other variables described in this manual, `load-path` cannot be changed via the Customize interface (see Section 33.1 [Easy Customization], page 686), but you can add a directory to it by putting a line like this in your init file (see Section 33.4 [Init File], page 711):

```
(add-to-list 'load-path "/path/to/my/lisp/library")
```

Some commands are *autoloaded*; when you run them, Emacs automatically loads the associated library first. For instance, the `M-x compile` command (see Section 24.1 [Compilation], page 534) is autoloaded; if you call it, Emacs automatically loads the `compile` library first. In contrast, the command `M-x recompile` is not autoloaded, so it is unavailable until you load the `compile` library.

Automatic loading can also occur when you look up the documentation of an autoloaded command (see ⟨undefined⟩ [Name Help], page ⟨undefined⟩), if the documentation refers to other functions and variables in its library (loading the library lets Emacs properly set up the hyperlinks in the '`*Help*`' buffer). To disable this feature, change the variable `help-enable-auto-load` to `nil`.

By default, Emacs refuses to load compiled Lisp files which were compiled with XEmacs, a modified versions of Emacs—they can cause Emacs to crash. Set the variable `load-dangerous-libraries` to `t` if you want to try loading them.

## 24.9 Evaluating Emacs Lisp Expressions

Emacs Lisp mode is the major mode for editing Emacs Lisp. Its mode command is `M-x emacs-lisp-mode`.

Emacs provides several commands for evaluating Emacs Lisp expressions. You can use these commands in Emacs Lisp mode, to test your Emacs Lisp code as it is being written. For example, after re-writing a function, you can evaluate the function definition to make it take effect for subsequent function calls. These commands are also available globally, and can be used outside Emacs Lisp mode.

`M-:` Read a single Emacs Lisp expression in the minibuffer, evaluate it, and print the value in the echo area (`eval-expression`).

`C-x C-e` Evaluate the Emacs Lisp expression before point, and print the value in the echo area (`eval-last-sexp`).

`C-M-x` (in Emacs Lisp mode)
`M-x eval-defun`
Evaluate the defun containing or after point, and print the value in the echo area (`eval-defun`).

`M-x eval-region`
Evaluate all the Emacs Lisp expressions in the region.

`M-x eval-buffer`
Evaluate all the Emacs Lisp expressions in the buffer.

`M-:` (`eval-expression`) reads an expression using the minibuffer, and evaluates it. (Before evaluating the expression, the current buffer switches back to the buffer that was current when you typed `M-:`, not the minibuffer into which you typed the expression.)

The command `C-x C-e` (`eval-last-sexp`) evaluates the Emacs Lisp expression preceding point in the buffer, and displays the value in the echo area. When the result of an evaluation is an integer, you can type `C-x C-e` a second time to display the value of the integer result in additional formats (octal, hexadecimal, and character).

If `M-:` or `C-x C-e` is given a prefix argument, it inserts the value into the current buffer at point, rather than displaying it in the echo area. The argument's value does not matter.

The `eval-defun` command is bound to `C-M-x` in Emacs Lisp mode. It evaluates the top-level Lisp expression containing or following point, and prints the value in the echo area. In this context, a top-level expression is referred to as a "defun", but it need not be an actual `defun` (function definition). In particular, this command treats `defvar` expressions specially. Normally, evaluating a `defvar` expression does nothing if the variable it defines already has a value. But this command unconditionally resets the variable to the initial value specified by the `defvar`; this is convenient for debugging Emacs Lisp programs. `defcustom` and `defface` expressions are treated similarly. Note that the other commands documented in this section do not have this special feature.

With a prefix argument, `C-M-x` instruments the function definition for Edebug, the Emacs Lisp Debugger. See Section "Instrumenting" in *the Emacs Lisp Reference Manual*.

The command `M-x eval-region` parses the text of the region as one or more Lisp expressions, evaluating them one by one. `M-x eval-buffer` is similar but evaluates the entire buffer.

The options `eval-expression-print-level` and `eval-expression-print-length` control the maximum depth and length of lists to print in the result of the evaluation commands before abbreviating them. `eval-expression-debug-on-error` controls

whether evaluation errors invoke the debugger when these commands are used; its default is `t`.

## 24.10 Lisp Interaction Buffers

When Emacs starts up, it contains a buffer named '`*scratch*`', which is provided for evaluating Emacs Lisp expressions interactively. Its major mode is Lisp Interaction mode. You can also enable Lisp Interaction mode by typing `M-x lisp-interaction-mode`.

In the '`*scratch*`' buffer, and other Lisp Interaction mode buffers, `C-j` (`eval-print-last-sexp`) evaluates the Lisp expression before point, and inserts the value at point. Thus, as you type expressions into the buffer followed by `C-j` after each expression, the buffer records a transcript of the evaluated expressions and their values. All other commands in Lisp Interaction mode are the same as in Emacs Lisp mode.

At startup, the '`*scratch*`' buffer contains a short message, in the form of a Lisp comment, that explains what it is for. This message is controlled by the variable `initial-scratch-message`, which should be either a string, or `nil` (which means to suppress the message).

An alternative way of evaluating Emacs Lisp expressions interactively is to use Inferior Emacs Lisp mode, which provides an interface rather like Shell mode (see Section 31.3.3 [Shell Mode], page 657) for evaluating Emacs Lisp expressions. Type `M-x ielm` to create an '`*ielm*`' buffer which uses this mode. For more information, see that command's documentation.

## 24.11 Running an External Lisp

Lisp mode is the major mode for editing programs written in general-purpose Lisp dialects, such as Common Lisp. Its mode command is `M-x lisp-mode`. Emacs uses Lisp mode automatically for files whose names end in '`.l`', '`.lsp`', or '`.lisp`'.

You can run an external Lisp session as a subprocess or *inferior process* of Emacs, and pass expressions to it to be evaluated. To begin an external Lisp session, type `M-x run-lisp`. This runs the program named `lisp`, and sets it up so that both input and output go through an Emacs buffer named '`*inferior-lisp*`'. To change the name of the Lisp program run by `M-x run-lisp`, change the variable `inferior-lisp-program`.

The major mode for the '`*lisp*`' buffer is Inferior Lisp mode, which combines the characteristics of Lisp mode and Shell mode (see Section 31.3.3 [Shell Mode], page 657). To send input to the Lisp session, go to the end of the '`*lisp*`' buffer and type the input, followed by RET. Terminal output from the Lisp session is automatically inserted in the buffer.

When you edit a Lisp program in Lisp mode, you can type `C-M-x` (`lisp-eval-defun`) to send an expression from the Lisp mode buffer to a Lisp session that you had started with `M-x run-lisp`. The expression sent is the top-level Lisp expression at or following point. The resulting value goes as usual into the '`*inferior-lisp*`' buffer. Note that the effect of `C-M-x` in Lisp mode is thus very similar to its effect in Emacs Lisp mode (see Section 24.9 [Lisp Eval], page 550), except that the expression is sent to a different Lisp environment instead of being evaluated in Emacs.

The facilities for editing Scheme code, and for sending expressions to a Scheme subprocess, are very similar. Scheme source files are edited in Scheme mode, which can be

explicitly enabled with `M-x scheme-mode`. You can initiate a Scheme session by typing `M-x run-scheme` (the buffer for interacting with Scheme is named '`*scheme*`'), and send expressions to it by typing `C-M-x`.

# 25 Maintaining Large Programs

This chapter describes Emacs features for maintaining large programs. If you are maintaining a large Lisp program, then in addition to the features described here, you may find the 'ERT' ("Emacs Lisp Regression Testing") library useful (see Section "ERT" in *Emacs Lisp Regression Testing*).

## 25.1 Version Control

A *version control system* is a program that can record multiple versions of a source file, storing information such as the creation time of each version, who made it, and a description of what was changed.

The Emacs version control interface is called *VC*. VC commands work with several different version control systems; currently, it supports GNU Arch, Bazaar, CVS, Git, Mercurial, Monotone, RCS, SCCS/CSSC, and Subversion. Of these, the GNU project distributes CVS, Arch, RCS, and Bazaar.

VC is enabled automatically whenever you visit a file governed by a version control system. To disable VC entirely, set the customizable variable `vc-handled-backends` to `nil` (see Section "Customizing VC" in *Specialized Emacs Features*).

### 25.1.1 Introduction to Version Control

VC allows you to use a version control system from within Emacs, integrating the version control operations smoothly with editing. It provides a uniform interface for common operations in many version control operations.

Some uncommon or intricate version control operations, such as altering repository settings, are not supported in VC. You should perform such tasks outside Emacs, e.g., via the command line.

This section provides a general overview of version control, and describes the version control systems that VC supports. You can skip this section if you are already familiar with the version control system you want to use.

#### 25.1.1.1 Understanding the problems it addresses

Version control systems provide you with three important capabilities:

- *Reversibility*: the ability to back up to a previous state if you discover that some modification you did was a mistake or a bad idea.
- *Concurrency*: the ability to have many people modifying the same collection of files knowing that conflicting modifications can be detected and resolved.
- *History*: the ability to attach historical data to your data, such as explanatory comments about the intention behind each change to it. Even for a programmer working solo, change histories are an important aid to memory; for a multi-person project, they are a vitally important form of communication among developers.

#### 25.1.1.2 Supported Version Control Systems

VC currently works with many different version control systems, which it refers to as *back ends*:

- SCCS was the first version control system ever built, and was long ago superseded by more advanced ones. VC compensates for certain features missing in SCCS (e.g., tag names for releases) by implementing them itself. Other VC features, such as multiple branches, are simply unavailable. Since SCCS is non-free, we recommend avoiding it.
- CSSC is a free replacement for SCCS. You should use CSSC only if, for some reason, you cannot use a more recent and better-designed version control system.
- RCS is the free version control system around which VC was initially built. It is relatively primitive: it cannot be used over the network, and works at the level of individual files. Almost everything you can do with RCS can be done through VC.
- CVS is the free version control system that was, until recently (circa 2008), used by the majority of free software projects. Nowadays, it is slowly being superseded by newer systems. CVS allows concurrent multi-user development either locally or over the network. Unlike newer systems, it lacks support for atomic commits and file moving/renaming. VC supports all basic editing operations under CVS.
- Subversion (svn) is a free version control system designed to be similar to CVS but without its problems (e.g., it supports atomic commits of filesets, and versioning of directories, symbolic links, meta-data, renames, copies, and deletes).
- GNU Arch is one of the earliest *decentralized* version control systems (the other being Monotone). See Section 25.1.1.3 [VCS Concepts], page 555, for a description of decentralized version control systems. It is no longer under active development, and has been deprecated in favor of Bazaar.
- Git is a decentralized version control system originally invented by Linus Torvalds to support development of Linux (his kernel). VC supports many common Git operations, but others, such as repository syncing, must be done from the command line.
- Mercurial (hg) is a decentralized version control system broadly resembling Git. VC supports most Mercurial commands, with the exception of repository sync operations.
- Bazaar (bzr) is a decentralized version control system that supports both repository-based and decentralized versioning. VC supports most basic editing operations under Bazaar.

### 25.1.1.3 Concepts of Version Control

When a file is under version control, we say that it is *registered* in the version control system. The system has a *repository* which stores both the file's present state and its change history—enough to reconstruct the current version or any earlier version. The repository also contains other information, such as *log entries* that describe the changes made to each file.

The copy of a version-controlled file that you actually edit is called the *work file*. You can change each work file as you would an ordinary file. After you are done with a set of changes, you may *commit* (or *check in*) the changes; this records the changes in the repository, along with a descriptive log entry.

A directory tree of work files is called a *working tree*.

Each commit creates a new *revision* in the repository. The version control system keeps track of all past revisions and the changes that were made in each revision. Each revision is named by a *revision ID*, whose format depends on the version control system; in the simplest case, it is just an integer.

To go beyond these basic concepts, you will need to understand three aspects in which version control systems differ. As explained in the next three sections, they can be lock-based or merge-based; file-based or changeset-based; and centralized or decentralized. VC handles all these modes of operation, but it cannot hide the differences.

#### 25.1.1.4 Merge-based vs lock-based Version Control

A version control system typically has some mechanism to coordinate between users who want to change the same file. There are two ways to do this: merging and locking.

In a version control system that uses merging, each user may modify a work file at any time. The system lets you *merge* your work file, which may contain changes that have not been committed, with the latest changes that others have committed.

Older version control systems use a *locking* scheme instead. Here, work files are normally read-only. To edit a file, you ask the version control system to make it writable for you by *locking* it; only one user can lock a given file at any given time. This procedure is analogous to, but different from, the locking that Emacs uses to detect simultaneous editing of ordinary files (see ⟨undefined⟩ [Interlocking], page ⟨undefined⟩). When you commit your changes, that unlocks the file, and the work file becomes read-only again. Other users may then lock the file to make their own changes.

Both locking and merging systems can have problems when multiple users try to modify the same file at the same time. Locking systems have *lock conflicts*; a user may try to check a file out and be unable to because it is locked. In merging systems, *merge conflicts* happen when you commit a change to a file that conflicts with a change committed by someone else after your checkout. Both kinds of conflict have to be resolved by human judgment and communication. Experience has shown that merging is superior to locking, both in convenience to developers and in minimizing the number and severity of conflicts that actually occur.

SCCS always uses locking. RCS is lock-based by default but can be told to operate in a merging style. CVS and Subversion are merge-based by default but can be told to operate in a locking mode. Decentralized version control systems, such as GNU Arch, Git, and Mercurial, are exclusively merging-based.

VC mode supports both locking and merging version control. The terms “commit” and “update” are used in newer version control systems; older lock-based systems use the terms “check in” and “check out”. VC hides the differences between them as much as possible.

#### 25.1.1.5 Changeset-based vs File-based Version Control

On SCCS, RCS, CVS, and other early version control systems, version control operations are *file-based*: each file has its own comment and revision history separate from that of all other files. Newer systems, beginning with Subversion, are *changeset-based*: a commit may include changes to several files, and the entire set of changes is handled as a unit. Any comment associated with the change does not belong to a single file, but to the changeset itself.

Changeset-based version control is more flexible and powerful than file-based version control; usually, when a change to multiple files has to be reversed, it’s good to be able to easily identify and remove all of it.

#### 25.1.1.6 Decentralized vs Centralized Repositories

Early version control systems were designed around a *centralized* model in which each project has only one repository used by all developers. SCCS, RCS, CVS, and Subversion share this kind of model. One of its drawbacks is that the repository is a choke point for reliability and efficiency.

GNU Arch pioneered the concept of *distributed* or *decentralized* version control, later implemented in Git, Mercurial, and Bazaar. A project may have several different repositories, and these systems support a sort of super-merge between repositories that tries to reconcile their change histories. In effect, there is one repository for each developer, and repository merges take the place of commit operations.

VC helps you manage the traffic between your personal workfiles and a repository. Whether the repository is a single master, or one of a network of peer repositories, is not something VC has to care about.

#### 25.1.1.7 Types of Log File

Projects that use a version control system can have two types of log for changes. One is the log maintained by the version control system: each time you commit a change, you fill out a *log entry* for the change (see Section 25.1.4 [Log Buffer], page 560). This is called the *version control log*.

The other kind of log is the file '`ChangeLog`' (see Section 25.2 [Change Log], page 570). It provides a chronological record of all changes to a large portion of a program—typically one directory and its subdirectories. A small program would use one '`ChangeLog`' file; a large program may have a '`ChangeLog`' file in each major directory. See Section 25.2 [Change Log], page 570. Programmers have used change logs since long before version control systems.

Changeset-based version systems typically maintain a changeset-based modification log for the entire system, which makes change log files somewhat redundant. One advantage that they retain is that it is sometimes useful to be able to view the transaction history of a single directory separately from those of other directories.

A project maintained with version control can use just the version control log, or it can use both kinds of logs. It can handle some files one way and some files the other way. Each project has its policy, which you should follow.

When the policy is to use both, you typically want to write an entry for each change just once, then put it into both logs. You can write the entry in '`ChangeLog`', then copy it to the log buffer with `C-c C-a` when committing the change (see Section 25.1.4 [Log Buffer], page 560). Or you can write the entry in the log buffer while committing the change, and later use the `C-x v a` command to copy it to '`ChangeLog`' (see Section "Change Logs and VC" in *Specialized Emacs Features*).

### 25.1.2 Version Control and the Mode Line

When you visit a file that is under version control, Emacs indicates this on the mode line. For example, '`Bzr-1223`' says that Bazaar is used for that file, and the current revision ID is 1223.

The character between the back-end name and the revision ID indicates the *version control status* of the work file. In a merge-based version control system, a '`-`' character indicates that the work file is unmodified, and '`:`' indicates that it has been modified. '`!`' indicates

that the file contains conflicts as result of a recent merge operation (see Section 25.1.10.3 [Merging], page 569), or that the file was removed from the version control. Finally, '`?`' means that the file is under version control, but is missing from the working tree.

In a lock-based system, '`-`' indicates an unlocked file, and '`:`' a locked file; if the file is locked by another user (for instance, '`jim`'), that is displayed as '`RCS:jim:1.3`'. '`@`' means that the file was locally added, but not yet committed to the master repository.

On a graphical display, you can move the mouse over this mode line indicator to pop up a "tool-tip", which displays a more verbose description of the version control status. Pressing `Mouse-1` over the indicator pops up a menu of VC commands, identical to '`Tools / Version Control`' on the menu bar.

When Auto Revert mode (see ⟨undefined⟩ [Reverting], page ⟨undefined⟩) reverts a buffer that is under version control, it updates the version control information in the mode line. However, Auto Revert mode may not properly update this information if the version control status changes without changes to the work file, from outside the current Emacs session. If you set `auto-revert-check-vc-info` to `t`, Auto Revert mode updates the version control status information every `auto-revert-interval` seconds, even if the work file itself is unchanged. The resulting CPU usage depends on the version control system, but is usually not excessive.

### 25.1.3 Basic Editing under Version Control

Most VC commands operate on *VC filesets*. A VC fileset is a collection of one or more files that a VC operation acts on. When you type VC commands in a buffer visiting a version-controlled file, the VC fileset is simply that one file. When you type them in a VC Directory buffer, and some files in it are marked, the VC fileset consists of the marked files (see Section 25.1.9 [VC Directory Mode], page 565).

On modern changeset-based version control systems (see Section 25.1.1.5 [VCS Changesets], page 556), VC commands handle multi-file VC filesets as a group. For example, committing a multi-file VC fileset generates a single revision, containing the changes to all those files. On older file-based version control systems like CVS, each file in a multi-file VC fileset is handled individually; for example, a commit generates one revision for each changed file.

`C-x v v` Perform the next appropriate version control operation on the current VC fileset.

The principal VC command is a multi-purpose command, `C-x v v` (`vc-next-action`), which performs the "most appropriate" action on the current VC fileset: either registering it with a version control system, or committing it, or unlocking it, or merging changes into it. The precise actions are described in detail in the following subsections. You can use `C-x v v` either in a file-visiting buffer or in a VC Directory buffer.

Note that VC filesets are distinct from the "named filesets" used for viewing and visiting files in functional groups (see ⟨undefined⟩ [Filesets], page ⟨undefined⟩). Unlike named filesets, VC filesets are not named and don't persist across sessions.

#### 25.1.3.1 Basic Version Control with Merging

On a merging-based version control system (i.e., most modern ones; see Section 25.1.1.4 [VCS Merging], page 556), `C-x v v` does the following:

- If there is more than one file in the VC fileset and the files have inconsistent version control statuses, signal an error. (Note, however, that a fileset is allowed to include both "newly-added" files and "modified" files; see Section 25.1.5 [Registering], page 561.)
- If none of the files in the VC fileset are registered with a version control system, register the VC fileset, i.e., place it under version control. See Section 25.1.5 [Registering], page 561. If Emacs cannot find a system to register under, it prompts for a repository type, creates a new repository, and registers the VC fileset with it.
- If every work file in the VC fileset is unchanged, do nothing.
- If every work file in the VC fileset has been modified, commit the changes. To do this, Emacs pops up a '`*vc-log*`' buffer; type the desired log entry for the new revision, followed by `C-c C-c` to commit. See Section 25.1.4 [Log Buffer], page 560.

  If committing to a shared repository, the commit may fail if the repository that has been changed since your last update. In that case, you must perform an update before trying again. On a decentralized version control system, use `C-x v +` (see Section 25.1.10.2 [VC Pull], page 568) or `C-x v m` (see Section 25.1.10.3 [Merging], page 569). On a centralized version control system, type `C-x v v` again to merge in the repository changes.
- Finally, if you are using a centralized version control system, check if each work file in the VC fileset is up-to-date. If any file has been changed in the repository, offer to update it.

These rules also apply when you use RCS in its "non-locking" mode, except that changes are not automatically merged from the repository. Nothing informs you if another user has committed changes in the same file since you began editing it; when you commit your revision, his changes are removed (however, they remain in the repository and are thus not irrevocably lost). Therefore, you must verify that the current revision is unchanged before committing your changes. In addition, locking is possible with RCS even in this mode: `C-x v v` with an unmodified file locks the file, just as it does with RCS in its normal locking mode (see Section 25.1.3.2 [VC With A Locking VCS], page 559).

### 25.1.3.2 Basic Version Control with Locking

On a locking-based version control system (such as SCCS, and RCS in its default mode), `C-x v v` does the following:

- If there is more than one file in the VC fileset and the files have inconsistent version control statuses, signal an error.
- If each file in the VC fileset is not registered with a version control system, register the VC fileset. See Section 25.1.5 [Registering], page 561. If Emacs cannot find a system to register under, it prompts for a repository type, creates a new repository, and registers the VC fileset with it.
- If each file is registered and unlocked, lock it and make it writable, so that you can begin to edit it.
- If each file is locked by you and contains changes, commit the changes. To do this, Emacs pops up a '`*vc-log*`' buffer; type the desired log entry for the new revision, followed by `C-c C-c` to commit (see Section 25.1.4 [Log Buffer], page 560).
- If each file is locked by you, but you have not changed it, release the lock and make the file read-only again.

- If each file is locked by another user, ask whether you want to "steal the lock". If you say yes, the file becomes locked by you, and a warning message is sent to the user who had formerly locked the file.

These rules also apply when you use CVS in locking mode, except that CVS does not support stealing locks.

#### 25.1.3.3 Advanced Control in `C-x v v`

When you give a prefix argument to `vc-next-action` (`C-u C-x v v`), it still performs the next logical version control operation, but accepts additional arguments to specify precisely how to do the operation.

- You can specify the name of a version control system. This is useful if the fileset can be managed by more than one version control system, and Emacs fails to detect the correct one.
- Otherwise, if using CVS or RCS, you can specify a revision ID.

  If the fileset is modified (or locked), this makes Emacs commit with that revision ID. You can create a new branch by supplying an appropriate revision ID (see Section 25.1.10 [Branches], page 567).

  If the fileset is unmodified (and unlocked), this checks the specified revision into the working tree. You can also specify a revision on another branch by giving its revision or branch ID (see Section 25.1.10.1 [Switching Branches], page 568). An empty argument (i.e., `C-u C-x v v RET`) checks out the latest ("head") revision on the current branch.

  This signals an error on a decentralized version control system. Those systems do not let you specify your own revision IDs, nor do they use the concept of "checking out" individual files.

### 25.1.4 Features of the Log Entry Buffer

When you tell VC to commit a change, it pops up a buffer named '`*vc-log*`'. In this buffer, you should write a *log entry* describing the changes you have made (see Section 25.1.1.1 [Why Version Control?], page 554). After you are done, type `C-c C-c` (`log-edit-done`) to exit the buffer and commit the change, together with your log entry.

The major mode for the '`*vc-log*`' buffer is Log Edit mode, a variant of Text mode (see ⟨undefined⟩ [Text Mode], page ⟨undefined⟩). On entering Log Edit mode, Emacs runs the hooks `text-mode-hook` and `vc-log-mode-hook` (see Section 33.2.2 [Hooks], page 696).

In the '`*vc-log*`' buffer, you can write one or more *header lines*, specifying additional information to be supplied to the version control system. Each header line must occupy a single line at the top of the buffer; the first line that is not a header line is treated as the start of the log entry. For example, the following header line states that the present change was not written by you, but by another developer:

```
Author: J. R. Hacker <jrh@example.com>
```

Apart from the '`Author`' header, Emacs recognizes the headers '`Date`' (a manually-specified commit time) and '`Fixes`' (a reference to a bug fixed by the change). Not all version control systems recognize all headers: Bazaar recognizes all three headers, while Git, Mercurial, and Monotone recognize only '`Author`' and '`Date`'. If you specify a header for a system that does not support it, the header is treated as part of the log entry.

While in the '*vc-log*' buffer, the "current VC fileset" is considered to be the fileset that will be committed if you type C-c C-c. To view a list of the files in the VC fileset, type C-c C-f (log-edit-show-files). To view a diff of changes between the VC fileset and the version from which you started editing (see Section 25.1.6 [Old Revisions], page 562), type C-c C-d (log-edit-show-diff).

If the VC fileset includes one or more 'ChangeLog' files (see Section 25.2 [Change Log], page 570), type C-c C-a (log-edit-insert-changelog) to pull the relevant entries into the '*vc-log*' buffer. If the topmost item in each 'ChangeLog' was made under your user name on the current date, this command searches that item for entries matching the file(s) to be committed, and inserts them.

To abort a commit, just **don't** type C-c C-c in that buffer. You can switch buffers and do other editing. As long as you don't try to make another commit, the entry you were editing remains in the '*vc-log*' buffer, and you can go back to that buffer at any time to complete the commit.

You can also browse the history of previous log entries to duplicate a commit comment. This can be useful when you want to make several commits with similar comments. The commands M-n, M-p, M-s and M-r for doing this work just like the minibuffer history commands (see Section 5.5 [Minibuffer History], page 74), except that they are used outside the minibuffer.

### 25.1.5 Registering a File for Version Control

C-x v i Register the visited file for version control.

The command C-x v i (vc-register) *registers* each file in the current VC fileset, placing it under version control. This is essentially equivalent to the action of C-x v v on an unregistered VC fileset (see Section 25.1.3 [Basic VC Editing], page 558), except that if the VC fileset is already registered, C-x v i signals an error whereas C-x v v performs some other action.

To register a file, Emacs must choose a version control system. For a multi-file VC fileset, the VC Directory buffer specifies the system to use (see Section 25.1.9 [VC Directory Mode], page 565). For a single-file VC fileset, if the file's directory already contains files registered in a version control system, or if the directory is part of a directory tree controlled by a version control system, Emacs chooses that system. In the event that more than one version control system is applicable, Emacs uses the one that appears first in the variable vc-handled-backends. If Emacs cannot find a version control system to register the file under, it prompts for a repository type, creates a new repository, and registers the file into that repository.

On most version control systems, registering a file with C-x v i or C-x v v adds it to the "working tree" but not to the repository. Such files are labeled as 'added' in the VC Directory buffer, and show a revision ID of '@@' in the mode line. To make the registration take effect in the repository, you must perform a commit (see Section 25.1.3 [Basic VC Editing], page 558). Note that a single commit can include both file additions and edits to existing files.

On a locking-based version control system (see Section 25.1.1.4 [VCS Merging], page 556), registering a file leaves it unlocked and read-only. Type C-x v v to start editing it.

### 25.1.6 Examining And Comparing Old Revisions

`C-x v =` Compare the work files in the current VC fileset with the versions you started from (`vc-diff`). With a prefix argument, prompt for two revisions of the current VC fileset and compare them. You can also call this command from a Dired buffer (see Chapter 27 [Dired], page 588).

`C-x v D` Compare the entire working tree to the revision you started from (`vc-root-diff`). With a prefix argument, prompt for two revisions and compare their trees.

`C-x v ~` Prompt for a revision of the current file, and visit it in a separate buffer (`vc-revision-other-window`).

`C-x v g` Display an annotated version of the current file: for each line, show the latest revision in which it was modified (`vc-annotate`).

`C-x v =` (`vc-diff`) displays a *diff* which compares each work file in the current VC fileset to the version(s) from which you started editing. The diff is displayed in another window, in a Diff mode buffer (see ⟨undefined⟩ [Diff Mode], page ⟨undefined⟩) named '`*vc-diff*`'. The usual Diff mode commands are available in this buffer. In particular, the `g` (`revert-buffer`) command performs the file comparison again, generating a new diff.

To compare two arbitrary revisions of the current VC fileset, call `vc-diff` with a prefix argument: `C-u C-x v =`. This prompts for two revision IDs (see Section 25.1.1.3 [VCS Concepts], page 555), and displays a diff between those versions of the fileset. This will not work reliably for multi-file VC filesets, if the version control system is file-based rather than changeset-based (e.g., CVS), since then revision IDs for different files would not be related in any meaningful way.

Instead of the revision ID, some version control systems let you specify revisions in other formats. For instance, under Bazaar you can enter '`date:yesterday`' for the argument to `C-u C-x v =` (and related commands) to specify the first revision committed after yesterday. See the documentation of the version control system for details.

If you invoke `C-x v =` or `C-u C-x v =` from a Dired buffer (see Chapter 27 [Dired], page 588), the file listed on the current line is treated as the current VC fileset.

`C-x v D` (`vc-root-diff`) is similar to `C-x v =`, but it displays the changes in the entire current working tree (i.e., the working tree containing the current VC fileset). If you invoke this command from a Dired buffer, it applies to the working tree containing the directory.

You can customize the `diff` options that `C-x v =` and `C-x v D` use for generating diffs. The options used are taken from the first non-`nil` value amongst the variables `vc-`*`backend`*`-diff-switches`, `vc-diff-switches`, and `diff-switches` (see ⟨undefined⟩ [Comparing Files], page ⟨undefined⟩), in that order. Here, *backend* stands for the relevant version control system, e.g., `bzr` for Bazaar. Since `nil` means to check the next variable in the sequence, either of the first two may use the value `t` to mean no switches at all. Most of the `vc-`*`backend`*`-diff-switches` variables default to `nil`, but some default to `t`; these are for version control systems whose `diff` implementations do not accept common diff options, such as Subversion.

To directly examine an older version of a file, visit the work file and type `C-x v ~` *`revision`* `RET` (`vc-revision-other-window`). This retrieves the file version corresponding to *revision*, saves it to '*`filename`*`.~`*`revision`*`~`', and visits it in a separate window.

Many version control systems allow you to view files *annotated* with per-line revision information, by typing `C-x v g` (`vc-annotate`). This creates a new buffer (the "annotate buffer") displaying the file's text, with each line colored to show how old it is. Red text is new, blue is old, and intermediate colors indicate intermediate ages. By default, the color is scaled over the full range of ages, such that the oldest changes are blue, and the newest changes are red.

When you give a prefix argument to this command, Emacs reads two arguments using the minibuffer: the revision to display and annotate (instead of the current file contents), and the time span in days the color range should cover.

From the annotate buffer, these and other color scaling options are available from the '`VC-Annotate`' menu. In this buffer, you can also use the following keys to browse the annotations of past revisions, view diffs, or view log entries:

`p` Annotate the previous revision, i.e., the revision before the one currently annotated. A numeric prefix argument is a repeat count, so `C-u 10 p` would take you back 10 revisions.

`n` Annotate the next revision, i.e., the revision after the one currently annotated. A numeric prefix argument is a repeat count.

`j` Annotate the revision indicated by the current line.

`a` Annotate the revision before the one indicated by the current line. This is useful to see the state the file was in before the change on the current line was made.

`f` Show in a buffer the file revision indicated by the current line.

`d` Display the diff between the current line's revision and the previous revision. This is useful to see what the current line's revision actually changed in the file.

`D` Display the diff between the current line's revision and the previous revision for all files in the changeset (for VC systems that support changesets). This is useful to see what the current line's revision actually changed in the tree.

`l` Show the log of the current line's revision. This is useful to see the author's description of the changes in the revision on the current line.

`w` Annotate the working revision–the one you are editing. If you used `p` and `n` to browse to other revisions, use this key to return to your working revision.

`v` Toggle the annotation visibility. This is useful for looking just at the file contents without distraction from the annotations.

### 25.1.7 VC Change Log

`C-x v l` Display the change history for the current fileset (`vc-print-log`).

`C-x v L` Display the change history for the current repository (`vc-print-root-log`).

`C-x v I` Display the changes that a pull operation will retrieve (`vc-log-incoming`).

`C-x v O` Display the changes that will be sent by the next push operation (`vc-log-outgoing`).

`C-x v l` (`vc-print-log`) displays a buffer named '`*vc-change-log*`', showing the history of changes made to the current file, including who made the changes, the dates, and the log entry for each change (these are the same log entries you would enter via the '`*vc-log*`' buffer; see Section 25.1.4 [Log Buffer], page 560). Point is centered at the revision of the file currently being visited. With a prefix argument, the command prompts for the revision to center on, and the maximum number of revisions to display.

If you call `C-x v l` from a VC Directory buffer (see Section 25.1.9 [VC Directory Mode], page 565) or a Dired buffer (see Chapter 27 [Dired], page 588), it applies to the file listed on the current line.

`C-x v L` (`vc-print-root-log`) displays a '`*vc-change-log*`' buffer showing the history of the entire version-controlled directory tree (RCS, SCCS, and CVS do not support this feature). With a prefix argument, the command prompts for the maximum number of revisions to display.

The `C-x v L` history is shown in a compact form, usually showing only the first line of each log entry. However, you can type RET (`log-view-toggle-entry-display`) in the '`*vc-change-log*`' buffer to reveal the entire log entry for the revision at point. A second RET hides it again.

On a decentralized version control system, the `C-x v I` (`vc-log-incoming`) command displays a log buffer showing the changes that will be applied, the next time you run the version control system's "pull" command to get new revisions from another repository (see Section 25.1.10.2 [VC Pull], page 568). This other repository is the default one from which changes are pulled, as defined by the version control system; with a prefix argument, `vc-log-incoming` prompts for a specific repository. Similarly, `C-x v O` (`vc-log-outgoing`) shows the changes that will be sent to another repository, the next time you run the "push" command; with a prefix argument, it prompts for a specific destination repository.

In the '`*vc-change-log*`' buffer, you can use the following keys to move between the logs of revisions and of files, and to examine and compare past revisions (see Section 25.1.6 [Old Revisions], page 562):

`p` Move to the previous revision entry. (Revision entries in the log buffer are usually in reverse-chronological order, so the previous revision-item usually corresponds to a newer revision.) A numeric prefix argument is a repeat count.

`n` Move to the next revision entry. A numeric prefix argument is a repeat count.

`P` Move to the log of the previous file, if showing logs for a multi-file VC fileset. Otherwise, just move to the beginning of the log. A numeric prefix argument is a repeat count.

`N` Move to the log of the next file, if showing logs for a multi-file VC fileset. A numeric prefix argument is a repeat count.

`a` Annotate the revision on the current line (see Section 25.1.6 [Old Revisions], page 562).

`e` Modify the change comment displayed at point. Note that not all VC systems support modifying change comments.

`f` Visit the revision indicated at the current line.

`d` Display a diff between the revision at point and the next earlier revision, for the specific file.

`D` Display the changeset diff between the revision at point and the next earlier revision. This shows the changes to all files made in that revision.

RET In a compact-style log buffer (e.g., the one created by `C-x v L`), toggle between showing and hiding the full log entry for the revision at point.

Because fetching many log entries can be slow, the '`*vc-change-log*`' buffer displays no more than 2000 revisions by default. The variable `vc-log-show-limit` specifies this limit; if you set the value to zero, that removes the limit. You can also increase the number of revisions shown in an existing '`*vc-change-log*`' buffer by clicking on the '`Show 2X entries`' or '`Show unlimited entries`' buttons at the end of the buffer. However, RCS, SCCS, and CVS do not support this feature.

### 25.1.8 Undoing Version Control Actions

`C-x v u` Revert the work file(s) in the current VC fileset to the last revision (`vc-revert`).

If you want to discard all the changes you have made to the current VC fileset, type `C-x v u` (`vc-revert-buffer`). This shows you a diff between the work file(s) and the revision from which you started editing, and asks for confirmation for discarding the changes. If you agree, the fileset is reverted. If you don't want `C-x v u` to show a diff, set the variable `vc-revert-show-diff` to `nil` (you can still view the diff directly with `C-x v =`; see Section 25.1.6 [Old Revisions], page 562). Note that `C-x v u` cannot be reversed with the usual undo commands (see Section 22.9 [Undo], page 469), so use it with care.

On locking-based version control systems, `C-x v u` leaves files unlocked; you must lock again to resume editing. You can also use `C-x v u` to unlock a file if you lock it and then decide not to change it.

### 25.1.9 VC Directory Mode

The *VC Directory buffer* is a specialized buffer for viewing the version control statuses of the files in a directory tree, and performing version control operations on those files. In particular, it is used to specify multi-file VC filesets for commands like `C-x v v` to act on (see Section 25.1.9.2 [VC Directory Commands], page 566).

To use the VC Directory buffer, type `C-x v d` (`vc-dir`). This reads a directory name using the minibuffer, and switches to a VC Directory buffer for that directory. By default, the buffer is named '`*vc-dir*`'. Its contents are described below.

The `vc-dir` command automatically detects the version control system to be used in the specified directory. In the event that more than one system is being used in the directory, you should invoke the command with a prefix argument, `C-u C-x v d`; this prompts for the version control system which the VC Directory buffer should use.

#### 25.1.9.1 The VC Directory Buffer

The VC Directory buffer contains a list of version-controlled files and their version control statuses. It lists files in the current directory (the one specified when you called `C-x v d`) and its subdirectories, but only those with a "noteworthy" status. Files that are up-to-date

(i.e., the same as in the repository) are omitted. If all the files in a subdirectory are up-to-date, the subdirectory is not listed either. As an exception, if a file has become up-to-date as a direct result of a VC command, it is listed.

Here is an example of a VC Directory buffer listing:

```
                    ./
    edited          configure.ac
*   added           README
    unregistered    temp.txt
                    src/
*   edited          src/main.c
```

Two work files have been modified but not committed: '`configure.ac`' in the current directory, and '`foo.c`' in the '`src/`' subdirectory. The file named '`README`' has been added but is not yet committed, while '`temp.txt`' is not under version control (see Section 25.1.5 [Registering], page 561).

The '`*`' characters next to the entries for '`README`' and '`src/main.c`' indicate that the user has marked out these files as the current VC fileset (see below).

The above example is typical for a decentralized version control system like Bazaar, Git, or Mercurial. Other systems can show other statuses. For instance, CVS shows the '`needs-update`' status if the repository has changes that have not been applied to the work file. RCS and SCCS show the name of the user locking a file as its status.

The VC Directory buffer omits subdirectories listed in the variable `vc-directory-exclusion-list`. Its default value contains directories that are used internally by version control systems.

### 25.1.9.2 VC Directory Commands

Emacs provides several commands for navigating the VC Directory buffer, and for "marking" files as belonging to the current VC fileset.

`n`
SPC — Move point to the next entry (`vc-dir-next-line`).

`p` — Move point to the previous entry (`vc-dir-previous-line`).

TAB — Move to the next directory entry (`vc-dir-next-directory`).

`S-TAB` — Move to the previous directory entry (`vc-dir-previous-directory`).

RET
`f` — Visit the file or directory listed on the current line (`vc-dir-find-file`).

`o` — Visit the file or directory on the current line, in a separate window (`vc-dir-find-file-other-window`).

`m` — Mark the file or directory on the current line (`vc-dir-mark`), putting it in the current VC fileset. If the region is active, mark all files in the region.

A file cannot be marked with this command if it is already in a marked directory, or one of its subdirectories. Similarly, a directory cannot be marked with this command if any file in its tree is marked.

`M` — If point is on a file entry, mark all files with the same status; if point is on a directory entry, mark all files in that directory tree (`vc-dir-mark-all-files`). With a prefix argument, mark all listed files and directories.

`q` Quit the VC Directory buffer, and bury it (`quit-window`).

`u` Unmark the file or directory on the current line. If the region is active, unmark all the files in the region (`vc-dir-unmark`).

`U` If point is on a file entry, unmark all files with the same status; if point is on a directory entry, unmark all files in that directory tree (`vc-dir-unmark-all-files`). With a prefix argument, unmark all files and directories.

`x` Hide files with '`up-to-date`' status (`vc-dir-hide-up-to-date`).

While in the VC Directory buffer, all the files that you mark with `m` (`vc-dir-mark`) or `M` (`vc-dir-mark`) are in the current VC fileset. If you mark a directory entry with `m`, all the listed files in that directory tree are in the current VC fileset. The files and directories that belong to the current VC fileset are indicated with a '`*`' character in the VC Directory buffer, next to their VC status. In this way, you can set up a multi-file VC fileset to be acted on by VC commands like `C-x v v` (see Section 25.1.3 [Basic VC Editing], page 558), `C-x v =` (see Section 25.1.6 [Old Revisions], page 562), and `C-x v u` (see Section 25.1.8 [VC Undo], page 565).

The VC Directory buffer also defines some single-key shortcuts for VC commands with the `C-x v` prefix: `=`, `+`, `l`, `i`, and `v`.

For example, you can commit a set of edited files by opening a VC Directory buffer, where the files are listed with the '`edited`' status; marking the files; and typing `v` or `C-x v v` (`vc-next-action`). If the version control system is changeset-based, Emacs will commit the files in a single revision.

While in the VC Directory buffer, you can also perform search and replace on the current VC fileset, with the following commands:

`S` Search the fileset (`vc-dir-search`).

`Q` Do a regular expression query replace on the fileset (`vc-dir-query-replace-regexp`).

`M-s a C-s` Do an incremental search on the fileset (`vc-dir-isearch`).

`M-s a C-M-s`
Do an incremental regular expression search on the fileset (`vc-dir-isearch-regexp`).

Apart from acting on multiple files, these commands behave much like their single-buffer counterparts (see Chapter 12 [Search], page 199).

The above commands are also available via the menu bar, and via a context menu invoked by `Mouse-2`. Furthermore, some VC backends use the menu to provide extra backend-specific commands. For example, Git and Bazaar allow you to manipulate *stashes* and *shelves* (where are a way to temporarily put aside uncommitted changes, and bring them back at a later time).

### 25.1.10 Version Control Branches

One use of version control is to support multiple independent lines of development, which are called *branches*. Branches are used for maintaining separate "stable" and "development" versions of a program, and for developing unrelated features in isolation from one another.

VC's support for branch operations is currently fairly limited. For decentralized version control systems, it provides commands for *updating* one branch with the contents of another, and for *merging* the changes made to two different branches (see Section 25.1.10.3 [Merging], page 569). For centralized version control systems, it supports checking out different branches and committing into new or different branches.

### 25.1.10.1 Switching between Branches

The various version control systems differ in how branches are implemented, and these differences cannot be entirely concealed by VC.

On some decentralized version control systems, including Bazaar and Mercurial in its normal mode of operation, each branch has its own working directory tree, so switching between branches just involves switching directories. On Git, switching between branches is done using the `git branch` command, which changes the contents of the working tree itself.

On centralized version control systems, you can switch between branches by typing `C-u C-x v v` in an up-to-date work file (see Section 25.1.3.3 [Advanced C-x v v], page 560), and entering the revision ID for a revision on another branch. On CVS, for instance, revisions on the *trunk* (the main line of development) normally have IDs of the form 1.1, 1.2, 1.3, . . . , while the first branch created from (say) revision 1.2 has revision IDs 1.2.1.1, 1.2.1.2, . . . , the second branch created from revision 1.2 has revision IDs 1.2.2.1, 1.2.2.2, . . . , and so forth. You can also specify the *branch ID*, which is a branch revision ID omitting its final component (e.g., 1.2.1), to switch to the latest revision on that branch.

On a locking-based system, switching to a different branch also unlocks (write-protects) the working tree.

Once you have switched to a branch, VC commands will apply to that branch until you switch away; for instance, any VC filesets that you commit will be committed to that specific branch.

### 25.1.10.2 Pulling Changes into a Branch

`C-x v +` On a decentralized version control system, update the current branch by "pulling in" changes from another location.

On a centralized version control system, update the current VC fileset.

On a decentralized version control system, the command `C-x v +` (`vc-pull`) updates the current branch and working tree. It is typically used to update a copy of a remote branch. If you supply a prefix argument, the command prompts for the exact version control command to use, which lets you specify where to pull changes from. Otherwise, it pulls from a default location determined by the version control system.

Amongst decentralized version control systems, `C-x v +` is currently supported only by Bazaar, Git, and Mercurial. On Bazaar, it calls `bzr pull` for ordinary branches (to pull from a master branch into a mirroring branch), and `bzr update` for a bound branch (to pull from a central repository). On Git, it calls `git pull` to fetch changes from a remote repository and merge it into the current branch. On Mercurial, it calls `hg pull -u` to fetch changesets from the default remote repository and update the working directory.

Prior to pulling, you can use `C-x v I` (`vc-log-incoming`) to view a log buffer of the changes to be applied. See Section 25.1.7 [VC Change Log], page 563.

On a centralized version control system like CVS, `C-x v +` updates the current VC fileset from the repository.

#### 25.1.10.3 Merging Branches

`C-x v m` On a decentralized version control system, merge changes from another branch into the current one.

On a centralized version control system, merge changes from another branch into the current VC fileset.

While developing a branch, you may sometimes need to *merge* in changes that have already been made in another branch. This is not a trivial operation, as overlapping changes may have been made to the two branches.

On a decentralized version control system, merging is done with the command `C-x v m` (`vc-merge`). On Bazaar, this prompts for the exact arguments to pass to `bzr merge`, offering a sensible default if possible. On Git, this prompts for the name of a branch to merge from, with completion (based on the branch names known to the current repository). The output from running the merge command is shown in a separate buffer.

On a centralized version control system like CVS, `C-x v m` prompts for a branch ID, or a pair of revision IDs (see Section 25.1.10.1 [Switching Branches], page 568); then it finds the changes from that branch, or the changes between the two revisions you specified, and merges those changes into the current VC fileset. If you just type RET, Emacs simply merges any changes that were made on the same branch since you checked the file out.

Immediately after performing a merge, only the working tree is modified, and you can review the changes produced by the merge with `C-x v D` and related commands (see Section 25.1.6 [Old Revisions], page 562). If the two branches contained overlapping changes, merging produces a *conflict*; a warning appears in the output of the merge command, and *conflict markers* are inserted into each affected work file, surrounding the two sets of conflicting changes. You must then resolve the conflict by editing the conflicted files. Once you are done, the modified files must be committed in the usual way for the merge to take effect (see Section 25.1.3 [Basic VC Editing], page 558).

#### 25.1.10.4 Creating New Branches

On centralized version control systems like CVS, Emacs supports creating new branches as part of a commit operation. When committing a modified VC fileset, type `C-u C-x v v` (`vc-next-action` with a prefix argument; see Section 25.1.3.3 [Advanced C-x v v], page 560). Then Emacs prompts for a revision ID for the new revision. You should specify a suitable branch ID for a branch starting at the current revision. For example, if the current revision is 2.5, the branch ID should be 2.5.1, 2.5.2, and so on, depending on the number of existing branches at that point.

To create a new branch at an older revision (one that is no longer the head of a branch), first select that revision (see Section 25.1.10.1 [Switching Branches], page 568). Your procedure will then differ depending on whether you are using a locking or merging-based VCS.

On a locking VCS, you will need to lock the old revision branch with `C-x v v`. You'll be asked to confirm, when you lock the old revision, that you really mean to create a new

branch—if you say no, you'll be offered a chance to lock the latest revision instead. On a merging-based VCS you will skip this step.

Then make your changes and type `C-x v v` again to commit a new revision. This creates a new branch starting from the selected revision.

After the branch is created, subsequent commits create new revisions on that branch. To leave the branch, you must explicitly select a different revision with `C-u C-x v v`.

## 25.2 Change Logs

Many software projects keep a *change log*. This is a file, normally named '`ChangeLog`', containing a chronological record of when and how the program was changed. Sometimes, there are several change log files, each recording the changes in one directory or directory tree.

### 25.2.1 Change Log Commands

The Emacs command `C-x 4 a` adds a new entry to the change log file for the file you are editing (`add-change-log-entry-other-window`). If that file is actually a backup file, it makes an entry appropriate for the file's parent—that is useful for making log entries for functions that have been deleted in the current version.

`C-x 4 a` visits the change log file and creates a new entry unless the most recent entry is for today's date and your name. It also creates a new item for the current file. For many languages, it can even guess the name of the function or other object that was changed.

When the variable `add-log-keep-changes-together` is non-`nil`, `C-x 4 a` adds to any existing item for the file rather than starting a new item.

You can combine multiple changes of the same nature. If you don't enter any text after the initial `C-x 4 a`, any subsequent `C-x 4 a` adds another symbol to the change log entry.

If `add-log-always-start-new-record` is non-`nil`, `C-x 4 a` always makes a new entry, even if the last entry was made by you and on the same date.

If the value of the variable `change-log-version-info-enabled` is non-`nil`, `C-x 4 a` adds the file's version number to the change log entry. It finds the version number by searching the first ten percent of the file, using regular expressions from the variable `change-log-version-number-regexp-list`.

The change log file is visited in Change Log mode. In this major mode, each bunch of grouped items counts as one paragraph, and each entry is considered a page. This facilitates editing the entries. `C-j` and auto-fill indent each new line like the previous line; this is convenient for entering the contents of an entry.

You can use the `next-error` command (by default bound to `C-x '`) to move between entries in the Change Log, when Change Log mode is on. You will jump to the actual site in the file that was changed, not just to the next Change Log entry. You can also use `previous-error` to move back in the same list.

You can use the command `M-x change-log-merge` to merge other log files into a buffer in Change Log Mode, preserving the date ordering of entries.

Version control systems are another way to keep track of changes in your program and keep a change log. In the VC log buffer, typing `C-c C-a` (`log-edit-insert-changelog`) inserts the relevant Change Log entry, if one exists. See Section 25.1.4 [Log Buffer], page 560.

### 25.2.2 Format of ChangeLog

A change log entry starts with a header line that contains the current date, your name (taken from the variable `add-log-full-name`), and your email address (taken from the variable `add-log-mailing-address`). Aside from these header lines, every line in the change log starts with a space or a tab. The bulk of the entry consists of *items*, each of which starts with a line starting with whitespace and a star. Here are two entries, both dated in May 1993, with two items and one item respectively.

```
1993-05-25  Richard Stallman  <rms@gnu.org>

        * man.el: Rename symbols ‘man-*’ to ‘Man-*’.
        (manual-entry): Make prompt string clearer.

        * simple.el (blink-matching-paren-distance):
        Change default to 12,000.

1993-05-24  Richard Stallman  <rms@gnu.org>

        * vc.el (minor-mode-map-alist): Don’t use it if it’s void.
        (vc-cancel-version): Doc fix.
```

One entry can describe several changes; each change should have its own item, or its own line in an item. Normally there should be a blank line between items. When items are related (parts of the same change, in different places), group them by leaving no blank line between them.

You should put a copyright notice and permission notice at the end of the change log file. Here is an example:

```
Copyright 1997, 1998 Free Software Foundation, Inc.
Copying and distribution of this file, with or without modification, are
permitted provided the copyright notice and this notice are preserved.
```

Of course, you should substitute the proper years and copyright holder.

## 25.3 Tags Tables

A *tag* is a reference to a subunit in a program or in a document. In source code, tags reference syntactic elements of the program: functions, subroutines, data types, macros, etc. In a document, tags reference chapters, sections, appendices, etc. Each tag specifies the name of the file where the corresponding subunit is defined, and the position of the subunit's definition in that file.

A *tags table* records the tags extracted by scanning the source code of a certain program or a certain document. Tags extracted from generated files reference the original files, rather than the generated files that were scanned during tag extraction. Examples of generated files include C files generated from Cweb source files, from a Yacc parser, or from Lex scanner definitions; ‘`.i`’ preprocessed C files; and Fortran files produced by preprocessing ‘`.fpp`’ source files.

To produce a tags table, you run the `etags` shell command on a document or the source code file. The ‘`etags`’ program writes the tags to a *tags table file*, or *tags file* in short. The conventional name for a tags file is ‘`TAGS`’. See Section 25.3.2 [Create Tags Table], page 574.

Emacs provides many commands for searching and replacing using the information recorded in tags tables. For instance, the `M-.` (`find-tag`) jumps to the location of a specified function definition in its source file. See Section 25.3.5 [Find Tag], page 577.

The Ebrowse facility is similar to `etags` but specifically tailored for C++. See Section "Ebrowse" in *Ebrowse User's Manual*. The Semantic package provides another way to generate and use tags, separate from the `etags` facility. See Section 23.10 [Semantic], page 529.

### 25.3.1 Source File Tag Syntax

Here is how tag syntax is defined for the most popular languages:

- In C code, any C function or typedef is a tag, and so are definitions of `struct`, `union` and `enum`. `#define` macro definitions, `#undef` and `enum` constants are also tags, unless you specify '`--no-defines`' when making the tags table. Similarly, global variables are tags, unless you specify '`--no-globals`', and so are struct members, unless you specify '`--no-members`'. Use of '`--no-globals`', '`--no-defines`' and '`--no-members`' can make the tags table file much smaller.

  You can tag function declarations and external variables in addition to function definitions by giving the '`--declarations`' option to `etags`.
- In C++ code, in addition to all the tag constructs of C code, member functions are also recognized; member variables are also recognized, unless you use the '`--no-members`' option. Tags for variables and functions in classes are named '*`class`*`::`*`variable`*' and '*`class`*`::`*`function`*'. `operator` definitions have tag names like '`operator+`'.
- In Java code, tags include all the constructs recognized in C++, plus the `interface`, `extends` and `implements` constructs. Tags for variables and functions in classes are named '*`class`*`.`*`variable`*' and '*`class`*`.`*`function`*'.
- In LaTeX documents, the arguments for `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\eqno`, `\label`, `\ref`, `\cite`, `\bibitem`, `\part`, `\appendix`, `\entry`, `\index`, `\def`, `\newcommand`, `\renewcommand`, `\newenvironment` and `\renewenvironment` are tags.

  Other commands can make tags as well, if you specify them in the environment variable `TEXTAGS` before invoking `etags`. The value of this environment variable should be a colon-separated list of command names. For example,

  ```
  TEXTAGS="mycommand:myothercommand"
  export TEXTAGS
  ```

  specifies (using Bourne shell syntax) that the commands '`\mycommand`' and '`\myothercommand`' also define tags.
- In Lisp code, any function defined with `defun`, any variable defined with `defvar` or `defconst`, and in general the first argument of any expression that starts with '`(def`' in column zero is a tag.
- In Scheme code, tags include anything defined with `def` or with a construct whose name starts with '`def`'. They also include variables set with `set!` at top level in the file.

Several other languages are also supported:

- In Ada code, functions, procedures, packages, tasks and types are tags. Use the '`--packages-only`' option to create tags for packages only.

  In Ada, the same name can be used for different kinds of entity (e.g., for a procedure and for a function). Also, for things like packages, procedures and functions, there is the spec (i.e., the interface) and the body (i.e., the implementation). To make it easier to pick the definition you want, Ada tag name have suffixes indicating the type of entity:

  '`/b`' package body.

  '`/f`' function.

  '`/k`' task.

  '`/p`' procedure.

  '`/s`' package spec.

  '`/t`' type.

  Thus, `M-x find-tag RET bidule/b RET` will go directly to the body of the package `bidule`, while `M-x find-tag RET bidule RET` will just search for any tag `bidule`.
- In assembler code, labels appearing at the start of a line, followed by a colon, are tags.
- In Bison or Yacc input files, each rule defines as a tag the nonterminal it constructs. The portions of the file that contain C code are parsed as C code.
- In Cobol code, tags are paragraph names; that is, any word starting in column 8 and followed by a period.
- In Erlang code, the tags are the functions, records and macros defined in the file.
- In Fortran code, functions, subroutines and block data are tags.
- In HTML input files, the tags are the `title` and the `h1`, `h2`, `h3` headers. Also, tags are `name=` in anchors and all occurrences of `id=`.
- In Lua input files, all functions are tags.
- In makefiles, targets are tags; additionally, variables are tags unless you specify '`--no-globals`'.
- In Objective C code, tags include Objective C definitions for classes, class categories, methods and protocols. Tags for variables and functions in classes are named '*`class`*`::`*`variable`*' and '*`class`*`::`*`function`*'.
- In Pascal code, the tags are the functions and procedures defined in the file.
- In Perl code, the tags are the packages, subroutines and variables defined by the `package`, `sub`, `my` and `local` keywords. Use '`--globals`' if you want to tag global variables. Tags for subroutines are named '*`package`*`::`*`sub`*'. The name for subroutines defined in the default package is '`main::`*`sub`*'.
- In PHP code, tags are functions, classes and defines. Vars are tags too, unless you use the '`--no-members`' option.
- In PostScript code, the tags are the functions.
- In Prolog code, tags are predicates and rules at the beginning of line.
- In Python code, `def` or `class` at the beginning of a line generate a tag.

You can also generate tags based on regexp matching (see Section 25.3.3 [Etags Regexps], page 575) to handle other formats and languages.

### 25.3.2 Creating Tags Tables

The `etags` program is used to create a tags table file. It knows the syntax of several languages, as described in the previous section. Here is how to run `etags`:

```
etags inputfiles...
```

The `etags` program reads the specified files, and writes a tags table named '`TAGS`' in the current working directory. You can optionally specify a different file name for the tags table by using the '`--output=file`' option; specifying '`-`' as a file name prints the tags table to standard output.

If the specified files don't exist, `etags` looks for compressed versions of them and uncompresses them to read them. Under MS-DOS, `etags` also looks for file names like '`mycode.cgz`' if it is given '`mycode.c`' on the command line and '`mycode.c`' does not exist.

If the tags table becomes outdated due to changes in the files described in it, you can update it by running the `etags` program again. If the tags table does not record a tag, or records it for the wrong file, then Emacs will not be able to find that definition until you update the tags table. But if the position recorded in the tags table becomes a little bit wrong (due to other editing), Emacs will still be able to find the right position, with a slight delay.

Thus, there is no need to update the tags table after each edit. You should update a tags table when you define new tags that you want to have listed, or when you move tag definitions from one file to another, or when changes become substantial.

You can make a tags table *include* another tags table, by passing the '`--include=file`' option to `etags`. It then covers all the files covered by the included tags file, as well as its own.

If you specify the source files with relative file names when you run `etags`, the tags file will contain file names relative to the directory where the tags file was initially written. This way, you can move an entire directory tree containing both the tags file and the source files, and the tags file will still refer correctly to the source files. If the tags file is '`-`' or is in the '`/dev`' directory, however, the file names are made relative to the current working directory. This is useful, for example, when writing the tags to '`/dev/stdout`'.

When using a relative file name, it should not be a symbolic link pointing to a tags file in a different directory, because this would generally render the file names invalid.

If you specify absolute file names as arguments to `etags`, then the tags file will contain absolute file names. This way, the tags file will still refer to the same files even if you move it, as long as the source files remain in the same place. Absolute file names start with '`/`', or with '*`device`*`:/`' on MS-DOS and MS-Windows.

When you want to make a tags table from a great number of files, you may have problems listing them on the command line, because some systems have a limit on its length. You can circumvent this limit by telling `etags` to read the file names from its standard input, by typing a dash in place of the file names, like this:

```
find . -name "*.[chCH]" -print | etags -
```

`etags` recognizes the language used in an input file based on its file name and contents. You can specify the language explicitly with the '`--language=name`' option. You can intermix these options with file names; each one applies to the file names that follow it. Specify '`--language=auto`' to tell `etags` to resume guessing the language from the file names and

file contents. Specify '`--language=none`' to turn off language-specific processing entirely; then `etags` recognizes tags by regexp matching alone (see Section 25.3.3 [Etags Regexps], page 575).

The option '`--parse-stdin=`*file*' is mostly useful when calling `etags` from programs. It can be used (only once) in place of a file name on the command line. `etags` will read from standard input and mark the produced tags as belonging to the file *file*.

'`etags --help`' outputs the list of the languages `etags` knows, and the file name rules for guessing the language. It also prints a list of all the available `etags` options, together with a short explanation. If followed by one or more '`--language=`*lang*' options, it outputs detailed information about how tags are generated for *lang*.

### 25.3.3 Etags Regexps

The '`--regex`' option to `etags` allows tags to be recognized by regular expression matching. You can intermix this option with file names; each one applies to the source files that follow it. If you specify multiple '`--regex`' options, all of them are used in parallel. The syntax is:

```
--regex=[{language}]/tagregexp/[nameregexp/]modifiers
```

The essential part of the option value is *tagregexp*, the regexp for matching tags. It is always used anchored, that is, it only matches at the beginning of a line. If you want to allow indented tags, use a regexp that matches initial whitespace; start it with '`[ \t]*`'.

In these regular expressions, '`\`' quotes the next character, and all the GCC character escape sequences are supported ('`\a`' for bell, '`\b`' for back space, '`\d`' for delete, '`\e`' for escape, '`\f`' for formfeed, '`\n`' for newline, '`\r`' for carriage return, '`\t`' for tab, and '`\v`' for vertical tab).

Ideally, *tagregexp* should not match more characters than are needed to recognize what you want to tag. If the syntax requires you to write *tagregexp* so it matches more characters beyond the tag itself, you should add a *nameregexp*, to pick out just the tag. This will enable Emacs to find tags more accurately and to do completion on tag names more reliably. You can find some examples below.

The *modifiers* are a sequence of zero or more characters that modify the way `etags` does the matching. A regexp with no modifiers is applied sequentially to each line of the input file, in a case-sensitive way. The modifiers and their meanings are:

'`i`' Ignore case when matching this regexp.

'`m`' Match this regular expression against the whole file, so that multi-line matches are possible.

'`s`' Match this regular expression against the whole file, and allow '`.`' in *tagregexp* to match newlines.

The '`-R`' option cancels all the regexps defined by preceding '`--regex`' options. It too applies to the file names following it. Here's an example:

```
etags --regex=/reg1/i voo.doo --regex=/reg2/m \
    bar.ber -R --lang=lisp los.er
```

Here `etags` chooses the parsing language for '`voo.doo`' and '`bar.ber`' according to their contents. `etags` also uses *reg1* to recognize additional tags in '`voo.doo`', and both *reg1* and *reg2* to recognize additional tags in '`bar.ber`'. *reg1* is checked against each line of

‘`voo.doo`’ and ‘`bar.ber`’, in a case-insensitive way, while *reg2* is checked against the whole ‘`bar.ber`’ file, permitting multi-line matches, in a case-sensitive way. `etags` uses only the Lisp tags rules, with no user-specified regexp matching, to recognize tags in ‘`los.er`’.

You can restrict a ‘`--regex`’ option to match only files of a given language by using the optional prefix `{`*language*`}`. (‘`etags --help`’ prints the list of languages recognized by `etags`.) This is particularly useful when storing many predefined regular expressions for `etags` in a file. The following example tags the `DEFVAR` macros in the Emacs source files, for the C language only:

```
--regex='{c}/[ \t]*DEFVAR_[A-Z_ \t(]+"\([^"]+\)"/'
```

When you have complex regular expressions, you can store the list of them in a file. The following option syntax instructs `etags` to read two files of regular expressions. The regular expressions contained in the second file are matched without regard to case.

```
--regex=@case-sensitive-file --ignore-case-regex=@ignore-case-file
```

A regex file for `etags` contains one regular expression per line. Empty lines, and lines beginning with space or tab are ignored. When the first character in a line is ‘`@`’, `etags` assumes that the rest of the line is the name of another file of regular expressions; thus, one such file can include another file. All the other lines are taken to be regular expressions. If the first non-whitespace text on the line is ‘`--`’, that line is a comment.

For example, we can create a file called ‘`emacs.tags`’ with the following contents:

```
      -- This is for GNU Emacs C source files
{c}/[ \t]*DEFVAR_[A-Z_ \t(]+"\([^"]+\)"/\1/
```

and then use it like this:

```
etags --regex=@emacs.tags *.[ch] */*.[ch]
```

Here are some more examples. The regexps are quoted to protect them from shell interpretation.

- Tag Octave files:

  ```
  etags --language=none \
        --regex='/[ \t]*function.*=[ \t]*\([^ \t]*\)[ \t]*(/\1/' \
        --regex='/###key \(.*\)/\1/' \
        --regex='/[ \t]*global[ \t].*/' \
        *.m
  ```

  Note that tags are not generated for scripts, so that you have to add a line by yourself of the form ‘`###key` *`scriptname`*’ if you want to jump to it.

- Tag Tcl files:

  ```
  etags --language=none --regex='/proc[ \t]+\([^ \t]+\)/\1/' *.tcl
  ```

- Tag VHDL files:

  ```
  etags --language=none \
    --regex='/[ \t]*\(ARCHITECTURE\|CONFIGURATION\) +[^ ]* +OF/' \
    --regex='/[ \t]*\(ATTRIBUTE\|ENTITY\|FUNCTION\|PACKAGE\
    \( BODY\)?\|PROCEDURE\|PROCESS\|TYPE\)[ \t]+\([^ \t(]+\)/\3/'
  ```

### 25.3.4 Selecting a Tags Table

Emacs has at any time one *selected* tags table. All the commands for working with tags tables use the selected one. To select a tags table, type `M-x visit-tags-table`, which reads the tags table file name as an argument, with ‘`TAGS`’ in the default directory as the default.

Emacs does not actually read in the tags table contents until you try to use them; all `visit-tags-table` does is store the file name in the variable `tags-file-name`, and setting the variable yourself is just as good. The variable's initial value is `nil`; that value tells all the commands for working with tags tables that they must ask for a tags table file name to use.

Using `visit-tags-table` when a tags table is already loaded gives you a choice: you can add the new tags table to the current list of tags tables, or start a new list. The tags commands use all the tags tables in the current list. If you start a new list, the new tags table is used *instead* of others. If you add the new table to the current list, it is used *as well as* the others.

You can specify a precise list of tags tables by setting the variable `tags-table-list` to a list of strings, like this:

```
(setq tags-table-list
      '("~/emacs" "/usr/local/lib/emacs/src"))
```

This tells the tags commands to look at the '`TAGS`' files in your '`~/emacs`' directory and in the '`/usr/local/lib/emacs/src`' directory. The order depends on which file you are in and which tags table mentions that file, as explained above.

Do not set both `tags-file-name` and `tags-table-list`.

### 25.3.5 Finding a Tag

The most important thing that a tags table enables you to do is to find the definition of a specific tag.

`M-.` *tag* `RET`
: Find first definition of *tag* (`find-tag`).

`C-u M-.`
: Find next alternate definition of last tag specified.

`C-u - M-.`
: Go back to previous tag found.

`C-M-.` *pattern* `RET`
: Find a tag whose name matches *pattern* (`find-tag-regexp`).

`C-u C-M-.`
: Find the next tag whose name matches the last pattern used.

`C-x 4 .` *tag* `RET`
: Find first definition of *tag*, but display it in another window (`find-tag-other-window`).

`C-x 5 .` *tag* `RET`
: Find first definition of *tag*, and create a new frame to select the buffer (`find-tag-other-frame`).

`M-*`
: Pop back to where you previously invoked `M-.` and friends.

`M-.` (`find-tag`) prompts for a tag name and jumps to its source definition. It works by searching through the tags table for that tag's file and approximate character position, visiting that file, and searching for the tag definition at ever-increasing distances away from the recorded approximate position.

When entering the tag argument to `M-.`, the usual minibuffer completion commands can be used (see Section 5.4 [Completion], page 70), with the tag names in the selected tags table

as completion candidates. If you specify an empty argument, the balanced expression in the buffer before or around point is the default argument. See Section 23.4.1 [Expressions], page 521.

You don't need to give `M-.` the full name of the tag; a part will do. `M-.` finds tags which contain that argument as a substring. However, it prefers an exact match to a substring match. To find other tags that match the same substring, give `find-tag` a numeric argument, as in `C-u M-.` or `M-0 M-.`; this does not read a tag name, but continues searching the tags table's text for another tag containing the same substring last used.

Like most commands that can switch buffers, `find-tag` has a variant that displays the new buffer in another window, and one that makes a new frame for it. The former is `C-x 4 .` (`find-tag-other-window`), and the latter is `C-x 5 .` (`find-tag-other-frame`).

To move back to previous tag definitions, use `C-u - M-.`; more generally, `M-.` with a negative numeric argument. Similarly, `C-x 4 .` with a negative argument finds the previous tag location in another window.

As well as going back to places you've found tags recently, you can go back to places *from where* you found them, using `M-*` (`pop-tag-mark`). Thus you can find and examine the definition of something with `M-.` and then return to where you were with `M-*`.

Both `C-u - M-.` and `M-*` allow you to retrace your steps to a depth determined by the variable `find-tag-marker-ring-length`.

The command `C-M-.` (`find-tag-regexp`) visits the tags that match a specified regular expression. It is just like `M-.` except that it does regexp matching instead of substring matching.

### 25.3.6 Searching and Replacing with Tags Tables

The commands in this section visit and search all the files listed in the selected tags table, one by one. For these commands, the tags table serves only to specify a sequence of files to search. These commands scan the list of tags tables starting with the first tags table (if any) that describes the current file, proceed from there to the end of the list, and then scan from the beginning of the list until they have covered all the tables in the list.

`M-x tags-search RET` *regexp* `RET`
: Search for *regexp* through the files in the selected tags table.

`M-x tags-query-replace RET` *regexp* `RET` *replacement* `RET`
: Perform a `query-replace-regexp` on each file in the selected tags table.

`M-,`
: Restart one of the commands above, from the current location of point (`tags-loop-continue`).

`M-x tags-search` reads a regexp using the minibuffer, then searches for matches in all the files in the selected tags table, one file at a time. It displays the name of the file being searched so you can follow its progress. As soon as it finds an occurrence, `tags-search` returns.

Having found one match, you probably want to find all the rest. Type `M-,` (`tags-loop-continue`) to resume the `tags-search`, finding one more match. This searches the rest of the current buffer, followed by the remaining files of the tags table.

`M-x tags-query-replace` performs a single `query-replace-regexp` through all the files in the tags table. It reads a regexp to search for and a string to replace with, just like ordinary `M-x query-replace-regexp`. It searches much like `M-x tags-search`, but repeatedly, processing matches according to your input. See Section 12.10.4 [Query Replace], page 213, for more information on query replace.

You can control the case-sensitivity of tags search commands by customizing the value of the variable `tags-case-fold-search`. The default is to use the same setting as the value of `case-fold-search` (see Section 12.9 [Search Case], page 211).

It is possible to get through all the files in the tags table with a single invocation of `M-x tags-query-replace`. But often it is useful to exit temporarily, which you can do with any input event that has no special query replace meaning. You can resume the query replace subsequently by typing `M-,`; this command resumes the last tags search or replace command that you did. For instance, to skip the rest of the current file, you can type `M-> M-,`.

The commands in this section carry out much broader searches than the `find-tag` family. The `find-tag` commands search only for definitions of tags that match your substring or regexp. The commands `tags-search` and `tags-query-replace` find every occurrence of the regexp, as ordinary search commands and replace commands do in the current buffer.

These commands create buffers only temporarily for the files that they have to search (those which are not already visited in Emacs buffers). Buffers in which no match is found are quickly killed; the others continue to exist.

As an alternative to `tags-search`, you can run `grep` as a subprocess and have Emacs show you the matching lines one by one. See Section 24.4 [Grep Searching], page 537.

### 25.3.7 Tags Table Inquiries

`C-M-i`
`M-TAB` Perform completion on the text around point, using the selected tags table if one is loaded (`completion-at-point`).

`M-x list-tags RET` *file* `RET`
Display a list of the tags defined in the program file *file*.

`M-x tags-apropos RET` *regexp* `RET`
Display a list of all tags matching *regexp*.

In most programming language modes, you can type `C-M-i` or `M-TAB` (`completion-at-point`) to complete the symbol at point. If there is a selected tags table, this command can use it to generate completion candidates. See Section 23.8 [Symbol Completion], page 528.

`M-x list-tags` reads the name of one of the files covered by the selected tags table, and displays a list of tags defined in that file. Do not include a directory as part of the file name unless the file name recorded in the tags table includes a directory.

`M-x tags-apropos` is like `apropos` for tags (see ⟨undefined⟩ [Apropos], page ⟨undefined⟩). It displays a list of tags in the selected tags table whose entries match *regexp*. If the variable `tags-apropos-verbose` is non-`nil`, it displays the names of the tags files together with the tag names. You can customize the appearance of the output by setting the variable `tags-tag-face` to a face. You can display additional output by customizing the variable `tags-apropos-additional-actions`; see its documentation for details.

`M-x next-file` visits files covered by the selected tags table. The first time it is called, it visits the first file covered by the table. Each subsequent call visits the next covered file, unless a prefix argument is supplied, in which case it returns to the first file.

## 25.4 Emacs Development Environment

EDE (*Emacs Development Environment*) is a package that simplifies the task of creating, building, and debugging large programs with Emacs. It provides some of the features of an IDE, or *Integrated Development Environment*, in Emacs.

This section provides a brief description of EDE usage. For full details on Ede, type `C-h i` and then select the EDE manual.

EDE is implemented as a global minor mode (see Section 20.3 [Minor Modes], page 413). To enable it, type `M-x global-ede-mode` or click on the '`Project Support (EDE)`' item in the '`Tools`' menu. You can also enable EDE each time you start Emacs, by adding the following line to your initialization file:

```
(global-ede-mode t)
```

Activating EDE adds a menu named '`Development`' to the menu bar. Many EDE commands, including the ones described below, can be invoked from this menu.

EDE organizes files into *projects*, which correspond to directory trees. The *project root* is the topmost directory of a project. To define a new project, visit a file in the desired project root and type `M-x ede-new`. This command prompts for a *project type*, which refers to the underlying method that EDE will use to manage the project (see Section "Creating a Project" in *Emacs Development Environment*). The most common project types are '`Make`', which uses Makefiles, and '`Automake`', which uses GNU Automake (see Section "Top" in *Automake*). In both cases, EDE also creates a file named '`Project.ede`', which stores information about the project.

A project may contain one or more *targets*. A target can be an object file, executable program, or some other type of file, which is "built" from one or more of the files in the project.

To add a new *target* to a project, type `C-c . t` (`M-x ede-new-target`). This command also asks if you wish to "add" the current file to that target, which means that the target is to be built from that file. After you have defined a target, you can add more files to it by typing `C-c . a` (`ede-add-file`).

To build a target, type `C-c . c` (`ede-compile-target`). To build all the targets in the project, type `C-c . C` (`ede-compile-project`). EDE uses the file types to guess how the target should be built.

# 26 Abbrevs and Abbrev Expansion

An abbreviation or *abbrev* is a string of characters that may be expanded to a longer string. The user can insert the abbrev string and find it replaced automatically with the expansion of the abbrev. This saves typing.

The set of abbrevs currently in effect is recorded in an *abbrev table*. Each buffer has a local abbrev table, but normally all buffers in the same major mode share one abbrev table. There is also a global abbrev table. Normally both are used.

An abbrev table is represented as an obarray. See ⟨undefined⟩ [Creating Symbols], page ⟨undefined⟩, for information about obarrays. Each abbreviation is represented by a symbol in the obarray. The symbol's name is the abbreviation; its value is the expansion; its function definition is the hook function for performing the expansion (see Section 26.2 [Defining Abbrevs], page 582); and its property list cell contains various additional properties, including the use count and the number of times the abbreviation has been expanded (see Section 26.6 [Abbrev Properties], page 586).

Certain abbrevs, called *system abbrevs*, are defined by a major mode instead of the user. A system abbrev is identified by its non-`nil` `:system` property (see Section 26.6 [Abbrev Properties], page 586). When abbrevs are saved to an abbrev file, system abbrevs are omitted. See Section 26.3 [Abbrev Files], page 583.

Because the symbols used for abbrevs are not interned in the usual obarray, they will never appear as the result of reading a Lisp expression; in fact, normally they are never used except by the code that handles abbrevs. Therefore, it is safe to use them in a nonstandard way.

If the minor mode Abbrev mode is enabled, the buffer-local variable `abbrev-mode` is non-`nil`, and abbrevs are automatically expanded in the buffer. For the user-level commands for abbrevs, see Section "Abbrev Mode" in *The GNU Emacs Manual*.

## 26.1 Abbrev Tables

This section describes how to create and manipulate abbrev tables.

`make-abbrev-table` **&optional** *props* [Function]
: This function creates and returns a new, empty abbrev table—an obarray containing no symbols. It is a vector filled with zeros. *props* is a property list that is applied to the new table (see Section 26.7 [Abbrev Table Properties], page 587).

`abbrev-table-p` *object* [Function]
: This function returns a non-`nil` value if *object* is an abbrev table.

`clear-abbrev-table` *abbrev-table* [Function]
: This function undefines all the abbrevs in *abbrev-table*, leaving it empty.

`copy-abbrev-table` *abbrev-table* [Function]
: This function returns a copy of *abbrev-table*—a new abbrev table containing the same abbrev definitions. It does *not* copy any property lists; only the names, values, and functions.

`define-abbrev-table` *tabname definitions* **&optional** *docstring* **&rest** *props* [Function]

This function defines *tabname* (a symbol) as an abbrev table name, i.e., as a variable whose value is an abbrev table. It defines abbrevs in the table according to *definitions*, a list of elements of the form `(abbrevname expansion [hook] [props...])`. These elements are passed as arguments to `define-abbrev`.

The optional string *docstring* is the documentation string of the variable *tabname*. The property list *props* is applied to the abbrev table (see Section 26.7 [Abbrev Table Properties], page 587).

If this function is called more than once for the same *tabname*, subsequent calls add the definitions in *definitions* to *tabname*, rather than overwriting the entire original contents. (A subsequent call only overrides abbrevs explicitly redefined or undefined in *definitions*.)

`abbrev-table-name-list` [Variable]

This is a list of symbols whose values are abbrev tables. `define-abbrev-table` adds the new abbrev table name to this list.

`insert-abbrev-table-description` *name* **&optional** *human* [Function]

This function inserts before point a description of the abbrev table named *name*. The argument *name* is a symbol whose value is an abbrev table.

If *human* is non-`nil`, the description is human-oriented. System abbrevs are listed and identified as such. Otherwise the description is a Lisp expression—a call to `define-abbrev-table` that would define *name* as it is currently defined, but without the system abbrevs. (The mode or package using *name* is supposed to add these to *name* separately.)

## 26.2 Defining Abbrevs

`define-abbrev` is the low-level basic function for defining an abbrev in an abbrev table.

When a major mode defines a system abbrev, it should call `define-abbrev` and specify `t` for the `:system` property. Be aware that any saved non-"system" abbrevs are restored at startup, i.e., before some major modes are loaded. Therefore, major modes should not assume that their abbrev tables are empty when they are first loaded.

`define-abbrev` *abbrev-table name expansion* **&optional** *hook* **&rest** *props* [Function]

This function defines an abbrev named *name*, in *abbrev-table*, to expand to *expansion* and call *hook*, with properties *props* (see Section 26.6 [Abbrev Properties], page 586). The return value is *name*. The `:system` property in *props* is treated specially here: if it has the value `force`, then it will overwrite an existing definition even for a non-"system" abbrev of the same name.

*name* should be a string. The argument *expansion* is normally the desired expansion (a string), or `nil` to undefine the abbrev. If it is anything but a string or `nil`, then the abbreviation "expands" solely by running *hook*.

The argument *hook* is a function or `nil`. If *hook* is non-`nil`, then it is called with no arguments after the abbrev is replaced with *expansion*; point is located at the end of *expansion* when *hook* is called.

If *hook* is a non-`nil` symbol whose `no-self-insert` property is non-`nil`, *hook* can explicitly control whether to insert the self-inserting input character that triggered the expansion. If *hook* returns non-`nil` in this case, that inhibits insertion of the character. By contrast, if *hook* returns `nil`, `expand-abbrev` (or `abbrev-insert`) also returns `nil`, as if expansion had not really occurred.

Normally, `define-abbrev` sets the variable `abbrevs-changed` to `t`, if it actually changes the abbrev. This is so that some commands will offer to save the abbrevs. It does not do this for a system abbrev, since those aren't saved anyway.

`only-global-abbrevs` [User Option]

If this variable is non-`nil`, it means that the user plans to use global abbrevs only. This tells the commands that define mode-specific abbrevs to define global ones instead. This variable does not alter the behavior of the functions in this section; it is examined by their callers.

## 26.3 Saving Abbrevs in Files

A file of saved abbrev definitions is actually a file of Lisp code. The abbrevs are saved in the form of a Lisp program to define the same abbrev tables with the same contents. Therefore, you can load the file with `load` (see ⟨undefined⟩ [How Programs Do Loading], page ⟨undefined⟩). However, the function `quietly-read-abbrev-file` is provided as a more convenient interface. Emacs automatically calls this function at startup.

User-level facilities such as `save-some-buffers` can save abbrevs in a file automatically, under the control of variables described here.

`abbrev-file-name` [User Option]

This is the default file name for reading and saving abbrevs.

`quietly-read-abbrev-file` **&optional** *filename* [Function]

This function reads abbrev definitions from a file named *filename*, previously written with `write-abbrev-file`. If *filename* is omitted or `nil`, the file specified in `abbrev-file-name` is used.

As the name implies, this function does not display any messages.

`save-abbrevs` [User Option]

A non-`nil` value for `save-abbrevs` means that Emacs should offer to save abbrevs (if any have changed) when files are saved. If the value is `silently`, Emacs saves the abbrevs without asking the user. `abbrev-file-name` specifies the file to save the abbrevs in.

`abbrevs-changed` [Variable]

This variable is set non-`nil` by defining or altering any abbrevs (except system abbrevs). This serves as a flag for various Emacs commands to offer to save your abbrevs.

`write-abbrev-file` **&optional** *filename* [Command]

Save all abbrev definitions (except system abbrevs), for all abbrev tables listed in `abbrev-table-name-list`, in the file *filename*, in the form of a Lisp program that when loaded will define the same abbrevs. If *filename* is `nil` or omitted, `abbrev-file-name` is used. This function returns `nil`.

## 26.4 Looking Up and Expanding Abbreviations

Abbrevs are usually expanded by certain interactive commands, including `self-insert-command`. This section describes the subroutines used in writing such commands, as well as the variables they use for communication.

`abbrev-symbol` *abbrev* **&optional** *table* [Function]
: This function returns the symbol representing the abbrev named *abbrev*. It returns `nil` if that abbrev is not defined. The optional second argument *table* is the abbrev table in which to look it up. If *table* is `nil`, this function tries first the current buffer's local abbrev table, and second the global abbrev table.

`abbrev-expansion` *abbrev* **&optional** *table* [Function]
: This function returns the string that *abbrev* would expand into (as defined by the abbrev tables used for the current buffer). It returns `nil` if *abbrev* is not a valid abbrev. The optional argument *table* specifies the abbrev table to use, as in `abbrev-symbol`.

`expand-abbrev` [Command]
: This command expands the abbrev before point, if any. If point does not follow an abbrev, this command does nothing. The command returns the abbrev symbol if it did expansion, `nil` otherwise.

  If the abbrev symbol has a hook function that is a symbol whose `no-self-insert` property is non-`nil`, and if the hook function returns `nil` as its value, then `expand-abbrev` returns `nil` even though expansion did occur.

`abbrev-insert` *abbrev* **&optional** *name start end* [Function]
: This function inserts the abbrev expansion of `abbrev`, replacing the text between `start` and `end`. If `start` is omitted, it defaults to point. `name`, if non-`nil`, should be the name by which this abbrev was found (a string); it is used to figure out whether to adjust the capitalization of the expansion. The function returns `abbrev` if the abbrev was successfully inserted.

`abbrev-prefix-mark` **&optional** *arg* [Command]
: This command marks the current location of point as the beginning of an abbrev. The next call to `expand-abbrev` will use the text from here to point (where it is then) as the abbrev to expand, rather than using the previous word as usual.

  First, this command expands any abbrev before point, unless *arg* is non-`nil`. (Interactively, *arg* is the prefix argument.) Then it inserts a hyphen before point, to indicate the start of the next abbrev to be expanded. The actual expansion removes the hyphen.

`abbrev-all-caps` [User Option]
: When this is set non-`nil`, an abbrev entered entirely in upper case is expanded using all upper case. Otherwise, an abbrev entered entirely in upper case is expanded by capitalizing each word of the expansion.

`abbrev-start-location` [Variable]
: The value of this variable is a buffer position (an integer or a marker) for `expand-abbrev` to use as the start of the next abbrev to be expanded. The value can also be

`nil`, which means to use the word before point instead. `abbrev-start-location` is set to `nil` each time `expand-abbrev` is called. This variable is also set by `abbrev-prefix-mark`.

`abbrev-start-location-buffer` [Variable]
The value of this variable is the buffer for which `abbrev-start-location` has been set. Trying to expand an abbrev in any other buffer clears `abbrev-start-location`. This variable is set by `abbrev-prefix-mark`.

`last-abbrev` [Variable]
This is the `abbrev-symbol` of the most recent abbrev expanded. This information is left by `expand-abbrev` for the sake of the `unexpand-abbrev` command (see Section "Expanding Abbrevs" in *The GNU Emacs Manual*).

`last-abbrev-location` [Variable]
This is the location of the most recent abbrev expanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

`last-abbrev-text` [Variable]
This is the exact expansion text of the most recent abbrev expanded, after case conversion (if any). Its value is `nil` if the abbrev has already been unexpanded. This contains information left by `expand-abbrev` for the sake of the `unexpand-abbrev` command.

`abbrev-expand-functions` [Variable]
This is a wrapper hook (see Section 20.1.1 [Running Hooks], page 396) run around the `expand-abbrev` function. Each function on this hook is called with a single argument: a function that performs the normal abbrev expansion. The hook function can hence do anything it wants before and after performing the expansion. It can also choose not to call its argument, thus overriding the default behavior; or it may even call it several times. The function should return the abbrev symbol if expansion took place.

The following sample code shows a simple use of `abbrev-expand-functions`. It assumes that `foo-mode` is a mode for editing certain files in which lines that start with '`#`' are comments. You want to use Text mode abbrevs for those lines. The regular local abbrev table, `foo-mode-abbrev-table` is appropriate for all other lines. See Section 26.5 [Standard Abbrev Tables], page 586, for the definitions of `local-abbrev-table` and `text-mode-abbrev-table`.

```
(defun foo-mode-abbrev-expand-function (expand)
  (if (not (save-excursion (forward-line 0) (eq (char-after) ?#)))
      ;; Performs normal expansion.
      (funcall expand)
    ;; We're inside a comment: use the text-mode abbrevs.
    (let ((local-abbrev-table text-mode-abbrev-table))
      (funcall expand))))

(add-hook 'foo-mode-hook
          #'(lambda ()
              (add-hook 'abbrev-expand-functions
                        'foo-mode-abbrev-expand-function
                        nil t)))
```

## 26.5 Standard Abbrev Tables

Here we list the variables that hold the abbrev tables for the preloaded major modes of Emacs.

`global-abbrev-table` [Variable]
: This is the abbrev table for mode-independent abbrevs. The abbrevs defined in it apply to all buffers. Each buffer may also have a local abbrev table, whose abbrev definitions take precedence over those in the global table.

`local-abbrev-table` [Variable]
: The value of this buffer-local variable is the (mode-specific) abbreviation table of the current buffer. It can also be a list of such tables.

`abbrev-minor-mode-table-alist` [Variable]
: The value of this variable is a list of elements of the form `(mode . abbrev-table)` where *mode* is the name of a variable: if the variable is bound to a non-`nil` value, then the *abbrev-table* is active, otherwise it is ignored. *abbrev-table* can also be a list of abbrev tables.

`fundamental-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Fundamental mode; in other words, it is the local abbrev table in all buffers in Fundamental mode.

`text-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Text mode.

`lisp-mode-abbrev-table` [Variable]
: This is the local abbrev table used in Lisp mode. It is the parent of the local abbrev table used in Emacs Lisp mode. See Section 26.7 [Abbrev Table Properties], page 587.

## 26.6 Abbrev Properties

Abbrevs have properties, some of which influence the way they work. You can provide them as arguments to `define-abbrev`, and manipulate them with the following functions:

`abbrev-put` *abbrev prop val* [Function]
: Set the property *prop* of *abbrev* to value *val*.

`abbrev-get` *abbrev prop* [Function]
: Return the property *prop* of *abbrev*, or `nil` if the abbrev has no such property.

The following properties have special meanings:

`:count`
: This property counts the number of times the abbrev has been expanded. If not explicitly set, it is initialized to 0 by `define-abbrev`.

`:system`
: If non-`nil`, this property marks the abbrev as a system abbrev. Such abbrevs are not saved (see Section 26.3 [Abbrev Files], page 583).

`:enable-function`
: If non-`nil`, this property should be a function of no arguments which returns `nil` if the abbrev should not be used and `t` otherwise.

`:case-fixed`
: If non-`nil`, this property indicates that the case of the abbrev's name is significant and should only match a text with the same pattern of capitalization. It also disables the code that modifies the capitalization of the expansion.

## 26.7 Abbrev Table Properties

Like abbrevs, abbrev tables have properties, some of which influence the way they work. You can provide them as arguments to `define-abbrev-table`, and manipulate them with the functions:

`abbrev-table-put` *table prop val* [Function]
: Set the property *prop* of abbrev table *table* to value *val*.

`abbrev-table-get` *table prop* [Function]
: Return the property *prop* of abbrev table *table*, or `nil` if the abbrev has no such property.

The following properties have special meaning:

`:enable-function`
: This is like the `:enable-function` abbrev property except that it applies to all abbrevs in the table. It is used before even trying to find the abbrev before point, so it can dynamically modify the abbrev table.

`:case-fixed`
: This is like the `:case-fixed` abbrev property except that it applies to all abbrevs in the table.

`:regexp` If non-`nil`, this property is a regular expression that indicates how to extract the name of the abbrev before point, before looking it up in the table. When the regular expression matches before point, the abbrev name is expected to be in submatch 1. If this property is `nil`, the default is to use `backward-word` and `forward-word` to find the name. This property allows the use of abbrevs whose name contains characters of non-word syntax.

`:parents` This property holds a list of tables from which to inherit other abbrevs.

`:abbrev-table-modiff`
: This property holds a counter incremented each time a new abbrev is added to the table.

# 27 Dired, the Directory Editor

Dired makes an Emacs buffer containing a listing of a directory, and optionally some of its subdirectories as well. You can use the normal Emacs commands to move around in this buffer, and special Dired commands to operate on the listed files.

The Dired buffer is "read-only", and inserting text in it is not allowed. Ordinary printing characters such as `d` and `x` are redefined for special Dired commands. Some Dired commands *mark* or *flag* the *current file* (that is, the file on the current line); other commands operate on the marked files or on the flagged files. You first mark certain files in order to operate on all of them with one command.

The Dired-X package provides various extra features for Dired mode. See Section "Top" in *Dired Extra User's Manual*.

You can also view a list of files in a directory with `C-x C-d` (`list-directory`). Unlike Dired, this command does not allow you to operate on the listed files. See ⟨undefined⟩ [Directories], page ⟨undefined⟩.

## 27.1 Entering Dired

To invoke Dired, type `C-x d` (`dired`). This reads a directory name using the minibuffer, and opens a *Dired buffer* listing the files in that directory. You can also supply a wildcard file name pattern as the minibuffer argument, in which case the Dired buffer lists all files matching that pattern. The usual history and completion commands can be used in the minibuffer; in particular, `M-n` puts the name of the visited file (if any) in the minibuffer (see Section 5.5 [Minibuffer History], page 74).

You can also invoke Dired by giving `C-x C-f` (`find-file`) a directory name.

The variable `dired-listing-switches` specifies the options to give to `ls` for listing the directory; this string *must* contain '`-l`'. If you use a prefix argument with the `dired` command, you can specify the `ls` switches with the minibuffer before you enter the directory specification. No matter how they are specified, the `ls` switches can include short options (that is, single characters) requiring no arguments, and long options (starting with '`--`') whose arguments are specified with '`=`'.

If your `ls` program supports the '`--dired`' option, Dired automatically passes it that option; this causes `ls` to emit special escape sequences for certain unusual file names, without which Dired will not be able to parse those names. The first time you run Dired in an Emacs session, it checks whether `ls` supports the '`--dired`' option by calling it once with that option. If the exit code is 0, Dired will subsequently use the '`--dired`' option; otherwise it will not. You can inhibit this check by customizing the variable `dired-use-ls-dired`. The value `unspecified` (the default) means to perform the check; any other non-`nil` value means to use the '`--dired`' option; and `nil` means not to use the '`--dired`' option.

On MS-Windows and MS-DOS systems, Emacs emulates `ls`. See Section I.4 [ls in Lisp], page 784, for options and peculiarities of this emulation.

To display the Dired buffer in another window, use `C-x 4 d` (`dired-other-window`). `C-x 5 d` (`dired-other-frame`) displays the Dired buffer in a separate frame.

Typing `q` (`quit-window`) buries the Dired buffer, and deletes its window if the window was created just for that buffer.

## 27.2 Navigation in the Dired Buffer

All the usual Emacs cursor motion commands are available in Dired buffers. The keys `C-n` and `C-p` are redefined to put the cursor at the beginning of the file name on the line, rather than at the beginning of the line.

For extra convenience, SPC and `n` in Dired are equivalent to `C-n`. `p` is equivalent to `C-p`. (Moving by lines is so common in Dired that it deserves to be easy to type.) DEL (move up and unflag) is also often useful simply for moving up (see Section 27.3 [Dired Deletion], page 589).

`j` (`dired-goto-file`) prompts for a file name using the minibuffer, and moves point to the line in the Dired buffer describing that file.

`M-s f C-s` (`dired-isearch-filenames`) performs a forward incremental search in the Dired buffer, looking for matches only amongst the file names and ignoring the rest of the text in the buffer. `M-s f M-C-s` (`dired-isearch-filenames-regexp`) does the same, using a regular expression search. If you change the variable `dired-isearch-filenames` to `t`, then the usual search commands also limit themselves to the file names; for instance, `C-s` behaves like `M-s f C-s`. If the value is `dwim`, then search commands match the file names only when point was on a file name initially. See Chapter 12 [Search], page 199, for information about incremental search.

Some additional navigation commands are available when the Dired buffer includes several directories. See Section 27.12 [Subdirectory Motion], page 598.

## 27.3 Deleting Files with Dired

One of the most frequent uses of Dired is to first *flag* files for deletion, then delete the files that were flagged.

`d` Flag this file for deletion (`dired-flag-file-deletion`).

`u` Remove the deletion flag (`dired-unmark`).

DEL Move point to previous line and remove the deletion flag on that line (`dired-unmark-backward`).

`x` Delete files flagged for deletion (`dired-do-flagged-delete`).

You can flag a file for deletion by moving to the line describing the file and typing `d` (`dired-flag-file-deletion`). The deletion flag is visible as a '`D`' at the beginning of the line. This command moves point to the next line, so that repeated `d` commands flag successive files. A numeric prefix argument serves as a repeat count; a negative count means to flag preceding files.

If the region is active, the `d` command flags all files in the region for deletion; in this case, the command does not move point, and ignores any prefix argument.

The reason for flagging files for deletion, rather than deleting files immediately, is to reduce the danger of deleting a file accidentally. Until you direct Dired to delete the flagged files, you can remove deletion flags using the commands `u` and DEL. `u` (`dired-unmark`) works just like `d`, but removes flags rather than making flags. DEL (`dired-unmark-backward`) moves upward, removing flags; it is like `u` with argument −1. A numeric prefix argument to either command serves as a repeat count, with a negative count meaning to

unflag in the opposite direction. If the region is active, these commands instead unflag all files in the region, without moving point.

To delete flagged files, type `x` (`dired-do-flagged-delete`). This command displays a list of all the file names flagged for deletion, and requests confirmation with `yes`. If you confirm, Dired deletes the flagged files, then deletes their lines from the text of the Dired buffer. The Dired buffer, with somewhat fewer lines, remains selected.

If you answer `no` or quit with `C-g` when asked to confirm, you return immediately to Dired, with the deletion flags still present in the buffer, and no files actually deleted.

You can delete empty directories just like other files, but normally Dired cannot delete directories that are nonempty. If the variable `dired-recursive-deletes` is non-`nil`, then Dired can delete nonempty directories including all their contents. That can be somewhat risky.

If you change the variable `delete-by-moving-to-trash` to `t`, the above deletion commands will move the affected files or directories into the operating system's Trash, instead of deleting them outright. See ⟨undefined⟩ [Misc File Ops], page ⟨undefined⟩.

## 27.4 Flagging Many Files at Once

The `#`, `~`, `.`, `% &`, and `% d` commands flag many files for deletion, based on their file names:

`#` Flag all auto-save files (files whose names start and end with '`#`') for deletion (see ⟨undefined⟩ [Auto Save], page ⟨undefined⟩).

`~` Flag all backup files (files whose names end with '`~`') for deletion (see ⟨undefined⟩ [Backup], page ⟨undefined⟩).

`.` (Period) Flag excess numeric backup files for deletion. The oldest and newest few backup files of any one file are exempt; the middle ones are flagged.

`% &` Flag for deletion all files with certain kinds of names which suggest you could easily create those files again.

`% d` *regexp* `RET`
Flag for deletion all files whose names match the regular expression *regexp*.

`#` (`dired-flag-auto-save-files`) flags all files whose names look like auto-save files—that is, files whose names begin and end with '`#`'. See ⟨undefined⟩ [Auto Save], page ⟨undefined⟩.

`~` (`dired-flag-backup-files`) flags all files whose names say they are backup files—that is, files whose names end in '`~`'. See ⟨undefined⟩ [Backup], page ⟨undefined⟩.

`.` (period, `dired-clean-directory`) flags just some of the backup files for deletion: all but the oldest few and newest few backups of any one file. Normally, the number of newest versions kept for each file is given by the variable `dired-kept-versions` (**not** `kept-new-versions`; that applies only when saving). The number of oldest versions to keep is given by the variable `kept-old-versions`.

Period with a positive numeric argument, as in `C-u 3 .`, specifies the number of newest versions to keep, overriding `dired-kept-versions`. A negative numeric argument overrides `kept-old-versions`, using minus the value of the argument to specify the number of oldest versions of each file to keep.

`% &` (`dired-flag-garbage-files`) flags files whose names match the regular expression specified by the variable `dired-garbage-files-regexp`. By default, this matches certain files produced by TEX, '`.bak`' files, and the '`.orig`' and '`.rej`' files produced by `patch`.

`% d` flags all files whose names match a specified regular expression (`dired-flag-files-regexp`). Only the non-directory part of the file name is used in matching. You can use '`^`' and '`$`' to anchor matches. You can exclude certain subdirectories from marking by hiding them while you use `% d`. See Section 27.13 [Hiding Subdirectories], page 599.

## 27.5 Visiting Files in Dired

There are several Dired commands for visiting or examining the files listed in the Dired buffer. All of them apply to the current line's file; if that file is really a directory, these commands invoke Dired on that subdirectory (making a separate Dired buffer).

`f` Visit the file described on the current line, like typing `C-x C-f` and supplying that file name (`dired-find-file`). See ⟨undefined⟩ [Visiting], page ⟨undefined⟩.

RET
`e` Equivalent to `f`.

`o` Like `f`, but uses another window to display the file's buffer (`dired-find-file-other-window`). The Dired buffer remains visible in the first window. This is like using `C-x 4 C-f` to visit the file. See Chapter 17 [Windows], page 289.

`C-o` Visit the file described on the current line, and display the buffer in another window, but do not select that window (`dired-display-file`).

`Mouse-1`
`Mouse-2` Visit the file whose name you clicked on (`dired-mouse-find-file-other-window`). This uses another window to display the file, like the `o` command.

`v` View the file described on the current line, with View mode (`dired-view-file`). View mode provides convenient commands to navigate the buffer but forbids changing it; See ⟨undefined⟩ [View Mode], page ⟨undefined⟩.

`^` Visit the parent directory of the current directory (`dired-up-directory`). This is equivalent to moving to the line for '`..`' and typing `f` there.

## 27.6 Dired Marks vs. Flags

Instead of flagging a file with '`D`', you can *mark* the file with some other character (usually '`*`'). Most Dired commands to operate on files use the files marked with '`*`'. The only command that operates on flagged files is `x`, which deletes them.

Here are some commands for marking with '`*`', for unmarking, and for operating on marks. (See Section 27.3 [Dired Deletion], page 589, for commands to flag and unflag files.)

`m`
`* m` Mark the current file with '`*`' (`dired-mark`). If the region is active, mark all files in the region instead; otherwise, if a numeric argument $n$ is supplied, mark the next $n$ files instead, starting with the current file (if $n$ is negative, mark the previous $-n$ files).

`* *` Mark all executable files with '`*`' (`dired-mark-executables`). With a numeric argument, unmark all those files.

`* @` Mark all symbolic links with '`*`' (`dired-mark-symlinks`). With a numeric argument, unmark all those files.

`* /` Mark with '`*`' all files which are directories, except for '`.`' and '`..`' (`dired-mark-directories`). With a numeric argument, unmark all those files.

`* s` Mark all the files in the current subdirectory, aside from '`.`' and '`..`' (`dired-mark-subdir-files`).

`u`
`* u` Remove any mark on this line (`dired-unmark`). If the region is active, unmark all files in the region instead; otherwise, if a numeric argument *n* is supplied, unmark the next *n* files instead, starting with the current file (if *n* is negative, unmark the previous $-n$ files).

DEL
`* DEL` Move point to previous line and remove any mark on that line (`dired-unmark-backward`). If the region is active, unmark all files in the region instead; otherwise, if a numeric argument *n* is supplied, unmark the *n* preceding files instead, starting with the current file (if *n* is negative, unmark the next $-n$ files).

`* !`
`U` Remove all marks from all the files in this Dired buffer (`dired-unmark-all-marks`).

`* ?` *markchar*
`M-DEL` Remove all marks that use the character *markchar* (`dired-unmark-all-files`). The argument is a single character—do not use RET to terminate it. See the description of the `* c` command below, which lets you replace one mark character with another.

With a numeric argument, this command queries about each marked file, asking whether to remove its mark. You can answer `y` meaning yes, `n` meaning no, or `!` to remove the marks from the remaining files without asking about them.

`* C-n`
`M-}` Move down to the next marked file (`dired-next-marked-file`) A file is "marked" if it has any kind of mark.

`* C-p`
`M-{` Move up to the previous marked file (`dired-prev-marked-file`)

`t`
`* t` Toggle all marks (`dired-toggle-marks`): files marked with '`*`' become unmarked, and unmarked files are marked with '`*`'. Files marked in any other way are not affected.

`* c` *old-markchar new-markchar*
Replace all marks that use the character *old-markchar* with marks that use the character *new-markchar* (`dired-change-marks`). This command is the

primary way to create or use marks other than '`*`' or '`D`'. The arguments are single characters—do not use RET to terminate them.

You can use almost any character as a mark character by means of this command, to distinguish various classes of files. If *old-markchar* is a space (' '), then the command operates on all unmarked files; if *new-markchar* is a space, then the command unmarks the files it acts on.

To illustrate the power of this command, here is how to put '`D`' flags on all the files that have no marks, while unflagging all those that already have '`D`' flags:

```
* c D t  * c SPC D  * c t SPC
```

This assumes that no files were already marked with '`t`'.

`% m` *regexp* `RET`
`* % ` *regexp* `RET`
: Mark (with '`*`') all files whose names match the regular expression *regexp* (`dired-mark-files-regexp`). This command is like `% d`, except that it marks files with '`*`' instead of flagging with '`D`'.

  Only the non-directory part of the file name is used in matching. Use '`^`' and '`$`' to anchor matches. You can exclude subdirectories by temporarily hiding them (see Section 27.13 [Hiding Subdirectories], page 599).

`% g` *regexp* `RET`
: Mark (with '`*`') all files whose *contents* contain a match for the regular expression *regexp* (`dired-mark-files-containing-regexp`). This command is like `% m`, except that it searches the file contents instead of the file name.

`C-/`
`C-x u`
`C-_`
: Undo changes in the Dired buffer, such as adding or removing marks (`dired-undo`). *This command does not revert the actual file operations, nor recover lost files!* It just undoes changes in the buffer itself.

  In some cases, using this after commands that operate on files can cause trouble. For example, after renaming one or more files, `dired-undo` restores the original names in the Dired buffer, which gets the Dired buffer out of sync with the actual contents of the directory.

## 27.7 Operating on Files

This section describes the basic Dired commands to operate on one file or several files. All of these commands are capital letters; all of them use the minibuffer, either to read an argument or to ask for confirmation, before they act. All of them let you specify the files to manipulate in these ways:

- If you give the command a numeric prefix argument $n$, it operates on the next $n$ files, starting with the current file. (If $n$ is negative, the command operates on the $-n$ files preceding the current line.)
- Otherwise, if some files are marked with '`*`', the command operates on all those files.
- Otherwise, the command operates on the current file only.

Certain other Dired commands, such as `!` and the '`%`' commands, use the same conventions to decide which files to work on.

Commands which ask for a destination directory, such as those which copy and rename files or create links for them, try to guess the default target directory for the operation. Normally, they suggest the Dired buffer's default directory, but if the variable `dired-dwim-target` is non-`nil`, and if there is another Dired buffer displayed in the next window, that other buffer's directory is suggested instead.

Here are the file-manipulating Dired commands that operate on files.

`C` *new* `RET` Copy the specified files (`dired-do-copy`). The argument *new* is the directory to copy into, or (if copying a single file) the new name. This is like the shell command `cp`.

If `dired-copy-preserve-time` is non-`nil`, then copying with this command preserves the modification time of the old file in the copy, like '`cp -p`'.

The variable `dired-recursive-copies` controls whether to copy directories recursively (like '`cp -r`'). The default is `top`, which means to ask before recursively copying a directory.

`D` Delete the specified files (`dired-do-delete`). This is like the shell command `rm`.

Like the other commands in this section, this command operates on the *marked* files, or the next *n* files. By contrast, `x` (`dired-do-flagged-delete`) deletes all *flagged* files.

`R` *new* `RET` Rename the specified files (`dired-do-rename`). If you rename a single file, the argument *new* is the new name of the file. If you rename several files, the argument *new* is the directory into which to move the files (this is like the shell command `mv`).

Dired automatically changes the visited file name of buffers associated with renamed files so that they refer to the new names.

`H` *new* `RET` Make hard links to the specified files (`dired-do-hardlink`). This is like the shell command `ln`. The argument *new* is the directory to make the links in, or (if making just one link) the name to give the link.

`S` *new* `RET` Make symbolic links to the specified files (`dired-do-symlink`). This is like '`ln -s`'. The argument *new* is the directory to make the links in, or (if making just one link) the name to give the link.

`M` *modespec* `RET`
Change the mode (also called *permission bits*) of the specified files (`dired-do-chmod`). *modespec* can be in octal or symbolic notation, like arguments handled by the `chmod` program.

`G` *newgroup* `RET`
Change the group of the specified files to *newgroup* (`dired-do-chgrp`).

`O` *newowner* `RET`
Change the owner of the specified files to *newowner* (`dired-do-chown`). (On most systems, only the superuser can do this.)

The variable `dired-chown-program` specifies the name of the program to use to do the work (different systems put `chown` in different places).

`T` *timestamp* `RET`
: Touch the specified files (`dired-do-touch`). This means updating their modification times to the present time. This is like the shell command `touch`.

`P` *command* `RET`
: Print the specified files (`dired-do-print`). You must specify the command to print them with, but the minibuffer starts out with a suitable guess made using the variables `lpr-command` and `lpr-switches` (the same variables that `lpr-buffer` uses; see Section 31.5 [Printing], page 669).

`Z`
: Compress the specified files (`dired-do-compress`). If the file appears to be a compressed file already, uncompress it instead.

`:d`
: Decrypt the specified files (`epa-dired-do-decrypt`). See Section "Dired integration" in *EasyPG Assistant User's Manual*.

`:v`
: Verify digital signatures on the specified files (`epa-dired-do-verify`). See Section "Dired integration" in *EasyPG Assistant User's Manual*.

`:s`
: Digitally sign the specified files (`epa-dired-do-sign`). See Section "Dired integration" in *EasyPG Assistant User's Manual*.

`:e`
: Encrypt the specified files (`epa-dired-do-encrypt`). See Section "Dired integration" in *EasyPG Assistant User's Manual*.

`L`
: Load the specified Emacs Lisp files (`dired-do-load`). See Section 24.8 [Lisp Libraries], page 549.

`B`
: Byte compile the specified Emacs Lisp files (`dired-do-byte-compile`). See Section "Byte Compilation" in *The Emacs Lisp Reference Manual*.

`A` *regexp* `RET`
: Search all the specified files for the regular expression *regexp* (`dired-do-search`).

  This command is a variant of `tags-search`. The search stops at the first match it finds; use `M-,` to resume the search and find the next match. See Section 25.3.6 [Tags Search], page 578.

`Q` *regexp* `RET` *to* `RET`
: Perform `query-replace-regexp` on each of the specified files, replacing matches for *regexp* with the string *to* (`dired-do-query-replace-regexp`).

  This command is a variant of `tags-query-replace`. If you exit the query replace loop, you can use `M-,` to resume the scan and replace more matches. See Section 25.3.6 [Tags Search], page 578.

## 27.8 Shell Commands in Dired

The Dired command `!` (`dired-do-shell-command`) reads a shell command string in the minibuffer, and runs that shell command on one or more files. The files that the shell

command operates on are determined in the usual way for Dired commands (see Section 27.7 [Operating on Files], page 593). The command `X` is a synonym for `!`.

The command `&` (`dired-do-async-shell-command`) does the same, except that it runs the shell command asynchronously. (You can also do this with `!`, by appending a '`&`' character to the end of the shell command.) When the command operates on more than one file, it runs multiple parallel copies of the specified shell command, one for each file. As an exception, if the specified shell command ends in '`;`' or '`;&`', the shell command is run in the background on each file sequentially; Emacs waits for each invoked shell command to terminate before running the next one.

For both `!` and `&`, the working directory for the shell command is the top-level directory of the Dired buffer.

If you tell `!` or `&` to operate on more than one file, the shell command string determines how those files are passed to the shell command:

- If you use '`*`' surrounded by whitespace in the command string, then the command runs just once, with the list of file names substituted for the '`*`'. The order of file names is the order of appearance in the Dired buffer.

  Thus, `! tar cf foo.tar * RET` runs `tar` on the entire list of file names, putting them into one tar file '`foo.tar`'.

  If you want to use '`*`' as a shell wildcard with whitespace around it, write '`*""`'. In the shell, this is equivalent to '`*`'; but since the '`*`' is not surrounded by whitespace, Dired does not treat it specially.
- Otherwise, if the command string contains '`?`' surrounded by whitespace, Emacs runs the shell command once *for each file*, substituting the current file name for '`?`' each time. You can use '`?`' more than once in the command; the same file name replaces each occurrence.
- If the command string contains neither '`*`' nor '`?`', Emacs runs the shell command once for each file, adding the file name at the end. For example, `! uudecode RET` runs `uudecode` on each file.

To iterate over the file names in a more complicated fashion, use an explicit shell loop. For example, here is how to uuencode each file, making the output file name by appending '`.uu`' to the input file name:

```
for file in * ; do uuencode "$file" "$file" >"$file".uu; done
```

The `!` and `&` commands do not attempt to update the Dired buffer to show new or modified files, because they don't know what files will be changed. Use the `g` command to update the Dired buffer (see Section 27.14 [Dired Updating], page 599).

See Section 31.3.1 [Single Shell], page 655, for information about running shell commands outside Dired.

## 27.9 Transforming File Names in Dired

This section describes Dired commands which alter file names in a systematic way. Each command operates on some or all of the marked files, using a new name made by transforming the existing name.

Like the basic Dired file-manipulation commands (see Section 27.7 [Operating on Files], page 593), the commands described here operate either on the next *n* files, or on all files

marked with '*', or on the current file. (To mark files, use the commands described in Section 27.6 [Marks vs Flags], page 591.)

All of the commands described in this section work *interactively*: they ask you to confirm the operation for each candidate file. Thus, you can select more files than you actually need to operate on (e.g., with a regexp that matches many files), and then filter the selected names by typing `y` or `n` when the command prompts for confirmation.

`% u` Rename each of the selected files to an upper-case name (`dired-upcase`). If the old file names are 'Foo' and 'bar', the new names are 'FOO' and 'BAR'.

`% l` Rename each of the selected files to a lower-case name (`dired-downcase`). If the old file names are 'Foo' and 'bar', the new names are 'foo' and 'bar'.

`% R` *from* `RET` *to* `RET`
`% C` *from* `RET` *to* `RET`
`% H` *from* `RET` *to* `RET`
`% S` *from* `RET` *to* `RET`
These four commands rename, copy, make hard links and make soft links, in each case computing the new name by regular-expression substitution from the name of the old file.

The four regular-expression substitution commands effectively perform a search-and-replace on the selected file names. They read two arguments: a regular expression *from*, and a substitution pattern *to*; they match each "old" file name against *from*, and then replace the matching part with *to*. You can use '`\&`' and '`\`*digit*' in *to* to refer to all or part of what the pattern matched in the old file name, as in `replace-regexp` (see Section 12.10.2 [Regexp Replace], page 212). If the regular expression matches more than once in a file name, only the first match is replaced.

For example, `% R ^.*$ RET x-\& RET` renames each selected file by prepending 'x-' to its name. The inverse of this, removing 'x-' from the front of each file name, is also possible: one method is `% R ^x-\(.*\)$ RET \1 RET`; another is `% R ^x- RET RET`. (Use '^' and '$' to anchor matches that should span the whole file name.)

Normally, the replacement process does not consider the files' directory names; it operates on the file name within the directory. If you specify a numeric argument of zero, then replacement affects the entire absolute file name including directory name. (A non-zero argument specifies the number of files to operate on.)

You may want to select the set of files to operate on using the same regexp *from* that you will use to operate on them. To do this, mark those files with `% m` *from* `RET`, then use the same regular expression in the command to operate on the files. To make this more convenient, the `%` commands to operate on files use the last regular expression specified in any `%` command as a default.

## 27.10 File Comparison with Dired

The `=` (`dired-diff`) command compares the current file (the file at point) with another file (read using the minibuffer) using the `diff` program. The file specified with the minibuffer is the first argument of `diff`, and file at point is the second argument. The output of the `diff` program is shown in a buffer using Diff mode (see ⟨undefined⟩ [Comparing Files], page ⟨undefined⟩).

If the region is active, the default for the file read using the minibuffer is the file at the mark (i.e., the ordinary Emacs mark, not a Dired mark; see Section 8.1 [Setting Mark], page 89). Otherwise, if the file at point has a backup file (see ⟨undefined⟩ [Backup], page ⟨undefined⟩), that is the default.

## 27.11 Subdirectories in Dired

A Dired buffer usually displays just one directory, but you can optionally include its subdirectories as well.

The simplest way to include multiple directories in one Dired buffer is to specify the options '`-lR`' for running `ls`. (If you give a numeric argument when you run Dired, then you can specify these options in the minibuffer.) That produces a recursive directory listing showing all subdirectories at all levels.

More often, you will want to show only specific subdirectories. You can do this with `i` (`dired-maybe-insert-subdir`):

`i` Insert the contents of a subdirectory later in the buffer.

If you use this command on a line that describes a file which is a directory, it inserts the contents of that directory into the same Dired buffer, and moves there. Inserted subdirectory contents follow the top-level directory of the Dired buffer, just as they do in '`ls -lR`' output.

If the subdirectory's contents are already present in the buffer, the `i` command just moves to it.

In either case, `i` sets the Emacs mark before moving, so `C-u C-SPC` returns to your previous position in the Dired buffer (see Section 8.1 [Setting Mark], page 89). You can also use '`^`' to return to the parent directory in the same Dired buffer (see Section 27.5 [Dired Visiting], page 591).

Use the `l` command (`dired-do-redisplay`) to update the subdirectory's contents, and use `C-u k` on the subdirectory header line to remove the subdirectory listing (see Section 27.14 [Dired Updating], page 599). You can also hide and show inserted subdirectories (see Section 27.13 [Hiding Subdirectories], page 599).

## 27.12 Moving Over Subdirectories

When a Dired buffer lists subdirectories, you can use the page motion commands `C-x [` and `C-x ]` to move by entire directories (see ⟨undefined⟩ [Pages], page ⟨undefined⟩).

The following commands move across, up and down in the tree of directories within one Dired buffer. They move to *directory header lines*, which are the lines that give a directory's name, at the beginning of the directory's contents.

`C-M-n` Go to next subdirectory header line, regardless of level (`dired-next-subdir`).

`C-M-p` Go to previous subdirectory header line, regardless of level (`dired-prev-subdir`).

`C-M-u` Go up to the parent directory's header line (`dired-tree-up`).

`C-M-d` Go down in the directory tree, to the first subdirectory's header line (`dired-tree-down`).

`<` Move up to the previous directory-file line (`dired-prev-dirline`). These lines are the ones that describe a directory as a file in its parent directory.

`>` Move down to the next directory-file line (`dired-prev-dirline`).

## 27.13 Hiding Subdirectories

*Hiding* a subdirectory means to make it invisible, except for its header line.

`$` Hide or show the subdirectory that point is in, and move point to the next subdirectory (`dired-hide-subdir`). This is a toggle. A numeric argument serves as a repeat count.

`M-$` Hide all subdirectories in this Dired buffer, leaving only their header lines (`dired-hide-all`). Or, if any subdirectory is currently hidden, make all subdirectories visible again. You can use this command to get an overview in very deep directory trees or to move quickly to subdirectories far away.

Ordinary Dired commands never consider files inside a hidden subdirectory. For example, the commands to operate on marked files ignore files in hidden directories even if they are marked. Thus you can use hiding to temporarily exclude subdirectories from operations without having to remove the Dired marks on files in those subdirectories.

See Section 27.14 [Dired Updating], page 599, for how to insert or delete a subdirectory listing.

## 27.14 Updating the Dired Buffer

This section describes commands to update the Dired buffer to reflect outside (non-Dired) changes in the directories and files, and to delete part of the Dired buffer.

`g` Update the entire contents of the Dired buffer (`revert-buffer`).

`l` Update the specified files (`dired-do-redisplay`). You specify the files for `l` in the same way as for file operations.

`k` Delete the specified *file lines*—not the files, just the lines (`dired-do-kill-lines`).

`s` Toggle between alphabetical order and date/time order (`dired-sort-toggle-or-edit`).

`C-u s` *switches* `RET`
Refresh the Dired buffer using *switches* as `dired-listing-switches`.

Type `g` (`revert-buffer`) to update the contents of the Dired buffer, based on changes in the files and directories listed. This preserves all marks except for those on files that have vanished. Hidden subdirectories are updated but remain hidden.

To update only some of the files, type `l` (`dired-do-redisplay`). Like the Dired file-operating commands, this command operates on the next $n$ files (or previous $-n$ files), or on the marked files if any, or on the current file. Updating the files means reading their current status, then updating their lines in the buffer to indicate that status.

If you use `l` on a subdirectory header line, it updates the contents of the corresponding subdirectory.

If you use `C-x d` or some other Dired command to visit a directory that is already being shown in a Dired buffer, Dired switches to that buffer but does not update it. If the buffer is not up-to-date, Dired displays a warning telling you to type G to update it. You can also tell Emacs to revert each Dired buffer automatically when you revisit it, by setting the variable `dired-auto-revert-buffer` to a non-`nil` value.

To delete *file lines* from the buffer—without actually deleting the files—type `k` (`dired-do-kill-lines`). Like the file-operating commands, this command operates on the next *n* files, or on the marked files if any. However, it does not operate on the current file, since otherwise mistyping `k` could be annoying.

If you use `k` to kill the line for a directory file which you had inserted in the Dired buffer as a subdirectory (see Section 27.11 [Subdirectories in Dired], page 598), it removes the subdirectory listing as well. Typing `C-u k` on the header line for a subdirectory also removes the subdirectory line from the Dired buffer.

The `g` command brings back any individual lines that you have killed in this way, but not subdirectories—you must use `i` to reinsert a subdirectory.

The files in a Dired buffers are normally listed in alphabetical order by file names. Alternatively Dired can sort them by date/time. The Dired command `s` (`dired-sort-toggle-or-edit`) switches between these two sorting modes. The mode line in a Dired buffer indicates which way it is currently sorted—by name, or by date.

`C-u s` *`switches`* `RET` lets you specify a new value for `dired-listing-switches`.

## 27.15 Dired and `find`

You can select a set of files for display in a Dired buffer more flexibly by using the `find` utility to choose the files.

To search for files with names matching a wildcard pattern use `M-x find-name-dired`. It reads arguments *directory* and *pattern*, and chooses all the files in *directory* or its subdirectories whose individual names match *pattern*.

The files thus chosen are displayed in a Dired buffer, in which the ordinary Dired commands are available.

If you want to test the contents of files, rather than their names, use `M-x find-grep-dired`. This command reads two minibuffer arguments, *directory* and *regexp*; it chooses all the files in *directory* or its subdirectories that contain a match for *regexp*. It works by running the programs `find` and `grep`. See also `M-x grep-find`, in Section 24.4 [Grep Searching], page 537. Remember to write the regular expression for `grep`, not for Emacs. (An alternative method of showing files whose contents match a given regexp is the `% g` *`regexp`* command, see Section 27.6 [Marks vs Flags], page 591.)

The most general command in this series is `M-x find-dired`, which lets you specify any condition that `find` can test. It takes two minibuffer arguments, *directory* and *find-args*; it runs `find` in *directory*, passing *find-args* to tell `find` what condition to test. To use this command, you need to know how to use `find`.

The format of listing produced by these commands is controlled by the variable `find-ls-option`. This is a pair of options; the first specifying how to call `find` to produce the file listing, and the second telling Dired to parse the output.

The command `M-x locate` provides a similar interface to the `locate` program. `M-x locate-with-filter` is similar, but keeps only files whose names match a given regular expression.

These buffers don't work entirely like ordinary Dired buffers: file operations work, but do not always automatically update the buffer. Reverting the buffer with `g` deletes all inserted subdirectories, and erases all flags and marks.

## 27.16 Editing the Dired Buffer

Wdired is a special mode that allows you to perform file operations by editing the Dired buffer directly (the "W" in "Wdired" stands for "writable".) To enter Wdired mode, type `C-x C-q` (`dired-toggle-read-only`) while in a Dired buffer. Alternatively, use the '`Immediate / Edit File Names`' menu item.

While in Wdired mode, you can rename files by editing the file names displayed in the Dired buffer. All the ordinary Emacs editing commands, including rectangle operations and `query-replace`, are available for this. Once you are done editing, type `C-c C-c` (`wdired-finish-edit`). This applies your changes and switches back to ordinary Dired mode.

Apart from simply renaming files, you can move a file to another directory by typing in the new file name (either absolute or relative). To mark a file for deletion, delete the entire file name. To change the target of a symbolic link, edit the link target name which appears next to the link name.

The rest of the text in the buffer, such as the file sizes and modification dates, is marked read-only, so you can't edit it. However, if you set `wdired-allow-to-change-permissions` to `t`, you can edit the file permissions. For example, you can change '`-rw-r--r--`' to '`-rw-rw-rw-`' to make a file world-writable. These changes also take effect when you type `C-c C-c`.

## 27.17 Viewing Image Thumbnails in Dired

Image-Dired is a facility for browsing image files. It provides viewing the images either as thumbnails or in full size, either inside Emacs or through an external viewer.

To enter Image-Dired, mark the image files you want to look at in the Dired buffer, using `m` as usual. Then type `C-t d` (`image-dired-display-thumbs`). This creates and switches to a buffer containing image-dired, corresponding to the marked files.

You can also enter Image-Dired directly by typing `M-x image-dired`. This prompts for a directory; specify one that has image files. This creates thumbnails for all the images in that directory, and displays them all in the "thumbnail buffer". This takes a long time if the directory contains many image files, and it asks for confirmation if the number of image files exceeds `image-dired-show-all-from-dir-max-files`.

With point in the thumbnail buffer, you can type `RET` (`image-dired-display-thumbnail-original-image`) to display a sized version of it in another window. This sizes the image to fit the window. Use the arrow keys to move around in the buffer. For easy browsing, use `SPC` (`image-dired-display-next-thumbnail-original`) to advance and display the next image. Typing `DEL` (`image-dired-display-previous-thumbnail-original`) backs up to the previous thumbnail and displays that instead.

To view and the image in its original size, either provide a prefix argument (`C-u`) before pressing `RET`, or type `C-RET` (`image-dired-thumbnail-display-external`) to display the image in an external viewer. You must first configure `image-dired-external-viewer`.

You can delete images through Image-Dired also. Type `d` (`image-dired-flag-thumb-original-file`) to flag the image file for deletion in the Dired buffer. You can also delete the thumbnail image from the thumbnail buffer with `C-d` (`image-dired-delete-char`).

More advanced features include *image tags*, which are metadata used to categorize image files. The tags are stored in a plain text file configured by `image-dired-db-file`.

To tag image files, mark them in the dired buffer (you can also mark files in Dired from the thumbnail buffer by typing `m`) and type `C-t t` (`image-dired-tag-files`). This reads the tag name in the minibuffer. To mark files having a certain tag, type `C-t f` (`image-dired-mark-tagged-files`). After marking image files with a certain tag, you can use `C-t d` to view them.

You can also tag a file directly from the thumbnail buffer by typing `t t` and you can remove a tag by typing `t r`. There is also a special "tag" called "comment" for each file (it is not a tag in the exact same sense as the other tags, it is handled slightly different). That is used to enter a comment or description about the image. You comment a file from the thumbnail buffer by typing `c`. You will be prompted for a comment. Type `C-t c` to add a comment from Dired (`image-dired-dired-comment-files`).

Image-Dired also provides simple image manipulation. In the thumbnail buffer, type `L` to rotate the original image 90 degrees anti clockwise, and `R` to rotate it 90 degrees clockwise. This rotation is lossless, and uses an external utility called JpegTRAN.

## 27.18 Other Dired Features

The command `+` (`dired-create-directory`) reads a directory name, and creates that directory. It signals an error if the directory already exists.

The command `M-s a C-s` (`dired-do-isearch`) begins a "multi-file" incremental search on the marked files. If a search fails at the end of a file, typing `C-s` advances to the next marked file and repeats the search; at the end of the last marked file, the search wraps around to the first marked file. The command `M-s a M-C-s` (`dired-do-isearch-regexp`) does the same with a regular expression search. See Section 12.1.2 [Repeat Isearch], page 200, for information about search repetition.

The command `w` (`dired-copy-filename-as-kill`) puts the names of the marked (or next *n*) files into the kill ring, as if you had killed them with `C-w`. The names are separated by a space.

With a zero prefix argument, this uses the absolute file name of each marked file. With just `C-u` as the prefix argument, it uses file names relative to the Dired buffer's default directory. (This can still contain slashes if in a subdirectory.) As a special case, if point is on a directory headerline, `w` gives you the absolute name of that directory. Any prefix argument or marked files are ignored in this case.

The main purpose of this command is so that you can yank the file names into arguments for other Emacs commands. It also displays what it added to the kill ring, so you can use it to display the list of currently marked files in the echo area.

If the directory you are visiting is under version control (see Section 25.1 [Version Control], page 554), then the normal VC diff and log commands will operate on the selected files.

The command `M-x dired-compare-directories` is used to compare the current Dired buffer with another directory. It marks all the files that are "different" between the two directories. It puts these marks in all Dired buffers where these files are listed, which of course includes the current buffer.

The default comparison method (used if you type RET at the prompt) is to compare just the file names—each file name that does not appear in the other directory is "different". You can specify more stringent comparisons by entering a Lisp expression, which can refer to the variables `size1` and `size2`, the respective file sizes; `mtime1` and `mtime2`, the last modification times in seconds, as floating point numbers; and `fa1` and `fa2`, the respective file attribute lists (as returned by the function `file-attributes`). This expression is evaluated for each pair of like-named files, and if the expression's value is non-`nil`, those files are considered "different".

For instance, the sequence `M-x dired-compare-directories RET (> mtime1 mtime2) RET` marks files newer in this directory than in the other, and marks files older in the other directory than in this one. It also marks files with no counterpart, in both directories, as always.

On the X Window System, Emacs supports the "drag and drop" protocol. You can drag a file object from another program, and drop it onto a Dired buffer; this either moves, copies, or creates a link to the file in that directory. Precisely which action is taken is determined by the originating program. Dragging files out of a Dired buffer is currently not supported.

# 28 The Calendar and the Diary

Emacs provides the functions of a desk calendar, with a diary of planned or past events. It also has facilities for managing your appointments, and keeping track of how much time you spend working on certain projects.

To enter the calendar, type `M-x calendar`; this displays a three-month calendar centered on the current month, with point on the current date. With a numeric argument, as in `C-u M-x calendar`, it prompts you for the month and year to be the center of the three-month calendar. The calendar uses its own buffer, whose major mode is Calendar mode.

`Mouse-3` in the calendar brings up a menu of operations on a particular date; `Mouse-2` brings up a menu of commonly used calendar features that are independent of any particular date. To exit the calendar, type `q`.

This chapter describes the basic calendar features. For more advanced topics, see Section "Advanced Calendar/Diary Usage" in *Specialized Emacs Features*.

## 28.1 Movement in the Calendar

Calendar mode provides commands to move through the calendar in logical units of time such as days, weeks, months, and years. If you move outside the three months originally displayed, the calendar display "scrolls" automatically through time to make the selected date visible. Moving to a date lets you view its holidays or diary entries, or convert it to other calendars; moving by long time periods is also useful simply to scroll the calendar.

### 28.1.1 Motion by Standard Lengths of Time

The commands for movement in the calendar buffer parallel the commands for movement in text. You can move forward and backward by days, weeks, months, and years.

`C-f` Move point one day forward (`calendar-forward-day`).

`C-b` Move point one day backward (`calendar-backward-day`).

`C-n` Move point one week forward (`calendar-forward-week`).

`C-p` Move point one week backward (`calendar-backward-week`).

`M-}` Move point one month forward (`calendar-forward-month`).

`M-{` Move point one month backward (`calendar-backward-month`).

`C-x ]` Move point one year forward (`calendar-forward-year`).

`C-x [` Move point one year backward (`calendar-backward-year`).

The day and week commands are natural analogues of the usual Emacs commands for moving by characters and by lines. Just as `C-n` usually moves to the same column in the following line, in Calendar mode it moves to the same day in the following week. And `C-p` moves to the same day in the previous week.

The arrow keys are equivalent to `C-f`, `C-b`, `C-n` and `C-p`, just as they normally are in other modes.

The commands for motion by months and years work like those for weeks, but move a larger distance. The month commands `M-}` and `M-{` move forward or backward by an entire month. The year commands `C-x ]` and `C-x [` move forward or backward a whole year.

The easiest way to remember these commands is to consider months and years analogous to paragraphs and pages of text, respectively. But the commands themselves are not quite analogous. The ordinary Emacs paragraph commands move to the beginning or end of a paragraph, whereas these month and year commands move by an entire month or an entire year, keeping the same date within the month or year.

All these commands accept a numeric argument as a repeat count. For convenience, the digit keys and the minus sign specify numeric arguments in Calendar mode even without the Meta modifier. For example, `100 C-f` moves point 100 days forward from its present location.

### 28.1.2 Beginning or End of Week, Month or Year

A week (or month, or year) is not just a quantity of days; we think of weeks (months, years) as starting on particular dates. So Calendar mode provides commands to move to the start or end of a week, month or year:

`C-a` Move point to start of week (`calendar-beginning-of-week`).

`C-e` Move point to end of week (`calendar-end-of-week`).

`M-a` Move point to start of month (`calendar-beginning-of-month`).

`M-e` Move point to end of month (`calendar-end-of-month`).

`M-<` Move point to start of year (`calendar-beginning-of-year`).

`M->` Move point to end of year (`calendar-end-of-year`).

These commands also take numeric arguments as repeat counts, with the repeat count indicating how many weeks, months, or years to move backward or forward.

By default, weeks begin on Sunday. To make them begin on Monday instead, set the variable `calendar-week-start-day` to 1.

### 28.1.3 Specified Dates

Calendar mode provides commands for moving to a particular date specified in various ways.

`g d` Move point to specified date (`calendar-goto-date`).

`g D` Move point to specified day of year (`calendar-goto-day-of-year`).

`g w` Move point to specified week of year (`calendar-iso-goto-week`).

`o` Center calendar around specified month (`calendar-other-month`).

`.` Move point to today's date (`calendar-goto-today`).

`g d` (`calendar-goto-date`) prompts for a year, a month, and a day of the month, and then moves to that date. Because the calendar includes all dates from the beginning of the current era, you must type the year in its entirety; that is, type '`1990`', not '`90`'.

`g D` (`calendar-goto-day-of-year`) prompts for a year and day number, and moves to that date. Negative day numbers count backward from the end of the year. `g w` (`calendar-iso-goto-week`) prompts for a year and week number, and moves to that week.

`o` (`calendar-other-month`) prompts for a month and year, then centers the three-month calendar around that month.

You can return to today's date with `.` (`calendar-goto-today`).

## 28.2 Scrolling in the Calendar

The calendar display scrolls automatically through time when you move out of the visible portion. You can also scroll it manually. Imagine that the calendar window contains a long strip of paper with the months on it. Scrolling the calendar means moving the strip horizontally, so that new months become visible in the window.

`>` Scroll calendar one month forward (`calendar-scroll-left`).

`<` Scroll calendar one month backward (`calendar-scroll-right`).

`C-v`
NEXT Scroll forward by three months (`calendar-scroll-left-three-months`).

`M-v`
PRIOR Scroll backward by three months (`calendar-scroll-right-three-months`).

The most basic calendar scroll commands scroll by one month at a time. This means that there are two months of overlap between the display before the command and the display after. `>` scrolls the calendar contents one month forward in time. `<` scrolls the contents one month backwards in time.

The commands `C-v` and `M-v` scroll the calendar by an entire "screenful"—three months—in analogy with the usual meaning of these commands. `C-v` makes later dates visible and `M-v` makes earlier dates visible. These commands take a numeric argument as a repeat count; in particular, since `C-u` multiplies the next command by four, typing `C-u C-v` scrolls the calendar forward by a year and typing `C-u M-v` scrolls the calendar backward by a year.

The function keys NEXT and PRIOR are equivalent to `C-v` and `M-v`, just as they are in other modes.

## 28.3 Counting Days

`M-=` Display the number of days in the current region (`calendar-count-days-region`).

To determine the number of days in a range, set the mark on one date using `C-SPC`, move point to another date, and type `M-=` (`calendar-count-days-region`). The numbers of days shown is *inclusive*; that is, it includes the days specified by mark and point.

## 28.4 Miscellaneous Calendar Commands

`p d` Display day-in-year (`calendar-print-day-of-year`).

`C-c C-l` Regenerate the calendar window (`calendar-redraw`).

`SPC` Scroll the next window up (`scroll-other-window`).

`DEL` Scroll the next window down (`scroll-other-window-down`).

`q` Exit from calendar (`calendar-exit`).

To display the number of days elapsed since the start of the year, or the number of days remaining in the year, type the `p d` command (`calendar-print-day-of-year`). This displays both of those numbers in the echo area. The count of days elapsed includes the selected date. The count of days remaining does not include that date.

If the calendar window text gets corrupted, type `C-c C-l` (`calendar-redraw`) to redraw it. (This can only happen if you use non-Calendar-mode editing commands.)

In Calendar mode, you can use `SPC` (`scroll-other-window`) and `DEL` (`scroll-other-window-down`) to scroll the other window (if there is one) up or down, respectively. This is handy when you display a list of holidays or diary entries in another window.

To exit from the calendar, type `q` (`calendar-exit`). This buries all buffers related to the calendar, selecting other buffers. (If a frame contains a dedicated calendar window, exiting from the calendar deletes or iconifies that frame depending on the value of `calendar-remove-frame-by-deleting`.)

## 28.5 Writing Calendar Files

You can write calendars and diary entries to HTML and LaTeX files.

The Calendar HTML commands produce files of HTML code that contain calendar, holiday, and diary entries. Each file applies to one month, and has a name of the format '*yyyy-mm*`.html`', where *yyyy* and *mm* are the four-digit year and two-digit month, respectively. The variable `cal-html-directory` specifies the default output directory for the HTML files. To prevent holidays from being shown, customize `cal-html-holidays`.

Diary entries enclosed by `<` and `>` are interpreted as HTML tags (for example: this is a diary entry with <font color="red">some red text</font>). You can change the overall appearance of the displayed HTML pages (for example, the color of various page elements, header styles) via a stylesheet '`cal.css`' in the directory containing the HTML files (see the value of the variable `cal-html-css-default` for relevant style settings).

`H m` Generate a one-month calendar (`cal-html-cursor-month`).

`H y` Generate a calendar file for each month of a year, as well as an index page (`cal-html-cursor-year`). By default, this command writes files to a *yyyy* subdirectory—if this is altered some hyperlinks between years will not work.

If the variable `cal-html-print-day-number-flag` is non-`nil`, then the monthly calendars show the day-of-the-year number. The variable `cal-html-year-index-cols` specifies the number of columns in the yearly index page.

The Calendar LaTeX commands produce a buffer of LaTeX code that prints as a calendar. Depending on the command you use, the printed calendar covers the day, week, month or year that point is in.

`t m` Generate a one-month calendar (`cal-tex-cursor-month`).

`t M` Generate a sideways-printing one-month calendar (`cal-tex-cursor-month-landscape`).

`t d` Generate a one-day calendar (`cal-tex-cursor-day`).

`t w 1` Generate a one-page calendar for one week, with hours (`cal-tex-cursor-week`).

`t w 2` Generate a two-page calendar for one week, with hours (`cal-tex-cursor-week2`).

`t w 3` Generate an ISO-style calendar for one week, without hours (`cal-tex-cursor-week-iso`).

`t w 4` Generate a calendar for one Monday-starting week, with hours (`cal-tex-cursor-week-monday`).

`t w W` Generate a two-page calendar for one week, without hours (`cal-tex-cursor-week2-summary`).

`t f w` Generate a Filofax-style two-weeks-at-a-glance calendar (`cal-tex-cursor-filofax-2week`).

`t f W` Generate a Filofax-style one-week-at-a-glance calendar (`cal-tex-cursor-filofax-week`).

`t y` Generate a calendar for one year (`cal-tex-cursor-year`).

`t Y` Generate a sideways-printing calendar for one year (`cal-tex-cursor-year-landscape`).

`t f y` Generate a Filofax-style calendar for one year (`cal-tex-cursor-filofax-year`).

Some of these commands print the calendar sideways (in "landscape mode"), so it can be wider than it is long. Some of them use Filofax paper size (3.75in x 6.75in). All of these commands accept a prefix argument, which specifies how many days, weeks, months or years to print (starting always with the selected one).

If the variable `cal-tex-holidays` is non-`nil` (the default), then the printed calendars show the holidays in `calendar-holidays`. If the variable `cal-tex-diary` is non-`nil` (the default is `nil`), diary entries are included also (in monthly, filofax, and iso-week calendars only). If the variable `cal-tex-rules` is non-`nil` (the default is `nil`), the calendar displays ruled pages in styles that have sufficient room. Consult the documentation of the individual cal-tex functions to see which calendars support which features.

You can use the variable `cal-tex-preamble-extra` to insert extra LaTeX commands in the preamble of the generated document if you need to.

## 28.6 Holidays

The Emacs calendar knows about many major and minor holidays, and can display them. You can add your own holidays to the default list.

`Mouse-3 Holidays`

`h` Display holidays for the selected date (`calendar-cursor-holidays`).

`x` Mark holidays in the calendar window (`calendar-mark-holidays`).

`u` Unmark calendar window (`calendar-unmark`).

`a` List all holidays for the displayed three months in another window (`calendar-list-holidays`).

`M-x holidays`

List all holidays for three months around today's date in another window.

`M-x list-holidays`

List holidays in another window for a specified range of years.

To see if any holidays fall on a given date, position point on that date in the calendar window and use the `h` command. Alternatively, click on that date with `Mouse-3` and then choose `Holidays` from the menu that appears. Either way, this displays the holidays for that date, in the echo area if they fit there, otherwise in a separate window.

To view the distribution of holidays for all the dates shown in the calendar, use the `x` command. This displays the dates that are holidays in a different face. See Section "Calendar Customizing" in *Specialized Emacs Features*. The command applies both to the currently visible months and to other months that subsequently become visible by scrolling. To turn marking off and erase the current marks, type `u`, which also erases any diary marks (see Section 28.10 [Diary], page 615). If the variable `calendar-mark-holidays-flag` is non-`nil`, creating or updating the calendar marks holidays automatically.

To get even more detailed information, use the `a` command, which displays a separate buffer containing a list of all holidays in the current three-month range. You can use SPC and DEL in the calendar window to scroll that list up and down, respectively.

The command `M-x holidays` displays the list of holidays for the current month and the preceding and succeeding months; this works even if you don't have a calendar window. If the variable `calendar-view-holidays-initially-flag` is non-`nil`, creating the calendar displays holidays in this way. If you want the list of holidays centered around a different month, use `C-u M-x holidays`, which prompts for the month and year.

The holidays known to Emacs include United States holidays and the major Bahá'í, Chinese, Christian, Islamic, and Jewish holidays; also the solstices and equinoxes.

The command `M-x holiday-list` displays the list of holidays for a range of years. This function asks you for the starting and stopping years, and allows you to choose all the holidays or one of several categories of holidays. You can use this command even if you don't have a calendar window.

The dates used by Emacs for holidays are based on *current practice*, not historical fact. For example Veteran's Day began in 1919, but is shown in earlier years.

## 28.7 Times of Sunrise and Sunset

Special calendar commands can tell you, to within a minute or two, the times of sunrise and sunset for any date.

`Mouse-3 Sunrise/sunset`

`S` Display times of sunrise and sunset for the selected date (`calendar-sunrise-sunset`).

`M-x sunrise-sunset`

Display times of sunrise and sunset for today's date.

`C-u M-x sunrise-sunset`

Display times of sunrise and sunset for a specified date.

`M-x calendar-sunrise-sunset-month`

Display times of sunrise and sunset for the selected month.

Within the calendar, to display the *local times* of sunrise and sunset in the echo area, move point to the date you want, and type `S`. Alternatively, click `Mouse-3` on the date, then choose '`Sunrise/sunset`' from the menu that appears. The command `M-x sunrise-sunset`

is available outside the calendar to display this information for today's date or a specified date. To specify a date other than today, use `C-u M-x sunrise-sunset`, which prompts for the year, month, and day.

You can display the times of sunrise and sunset for any location and any date with `C-u C-u M-x sunrise-sunset`. This asks you for a longitude, latitude, number of minutes difference from Coordinated Universal Time, and date, and then tells you the times of sunrise and sunset for that location on that date.

Because the times of sunrise and sunset depend on the location on earth, you need to tell Emacs your latitude, longitude, and location name before using these commands. Here is an example of what to set:

```
(setq calendar-latitude 40.1)
(setq calendar-longitude -88.2)
(setq calendar-location-name "Urbana, IL")
```

Use one decimal place in the values of `calendar-latitude` and `calendar-longitude`.

Your time zone also affects the local time of sunrise and sunset. Emacs usually gets time zone information from the operating system, but if these values are not what you want (or if the operating system does not supply them), you must set them yourself. Here is an example:

```
(setq calendar-time-zone -360)
(setq calendar-standard-time-zone-name "CST")
(setq calendar-daylight-time-zone-name "CDT")
```

The value of `calendar-time-zone` is the number of minutes difference between your local standard time and Coordinated Universal Time (Greenwich time). The values of `calendar-standard-time-zone-name` and `calendar-daylight-time-zone-name` are the abbreviations used in your time zone. Emacs displays the times of sunrise and sunset *corrected for daylight saving time*. See Section 28.13 [Daylight Saving], page 621, for how daylight saving time is determined.

As a user, you might find it convenient to set the calendar location variables for your usual physical location in your '`.emacs`' file. If you are a system administrator, you may want to set these variables for all users in a '`default.el`' file. See Section 33.4 [Init File], page 711.

## 28.8 Phases of the Moon

These calendar commands display the dates and times of the phases of the moon (new moon, first quarter, full moon, last quarter). This feature is useful for debugging problems that "depend on the phase of the moon".

`M` Display the dates and times for all the quarters of the moon for the three-month period shown (`calendar-lunar-phases`).

`M-x lunar-phases`
Display dates and times of the quarters of the moon for three months around today's date.

Within the calendar, use the `M` command to display a separate buffer of the phases of the moon for the current three-month range. The dates and times listed are accurate to within a few minutes.

Outside the calendar, use the command `M-x lunar-phases` to display the list of the phases of the moon for the current month and the preceding and succeeding months. For information about a different month, use `C-u M-x lunar-phases`, which prompts for the month and year.

The dates and times given for the phases of the moon are given in local time (corrected for daylight saving, when appropriate). See the discussion in the previous section. See Section 28.7 [Sunrise/Sunset], page 609.

## 28.9 Conversion To and From Other Calendars

The Emacs calendar displayed is *always* the Gregorian calendar, sometimes called the "new style" calendar, which is used in most of the world today. However, this calendar did not exist before the sixteenth century and was not widely used before the eighteenth century; it did not fully displace the Julian calendar and gain universal acceptance until the early twentieth century. The Emacs calendar can display any month since January, year 1 of the current era, but the calendar displayed is always the Gregorian, even for a date at which the Gregorian calendar did not exist.

While Emacs cannot display other calendars, it can convert dates to and from several other calendars.

### 28.9.1 Supported Calendar Systems

The ISO commercial calendar is often used in business.

The Julian calendar, named after Julius Caesar, was the one used in Europe throughout medieval times, and in many countries up until the nineteenth century.

Astronomers use a simple counting of days elapsed since noon, Monday, January 1, 4713 B.C. on the Julian calendar. The number of days elapsed is called the *Julian day number* or the *Astronomical day number*.

The Hebrew calendar is used by tradition in the Jewish religion. The Emacs calendar program uses the Hebrew calendar to determine the dates of Jewish holidays. Hebrew calendar dates begin and end at sunset.

The Islamic calendar is used in many predominantly Islamic countries. Emacs uses it to determine the dates of Islamic holidays. There is no universal agreement in the Islamic world about the calendar; Emacs uses a widely accepted version, but the precise dates of Islamic holidays often depend on proclamation by religious authorities, not on calculations. As a consequence, the actual dates of observance can vary slightly from the dates computed by Emacs. Islamic calendar dates begin and end at sunset.

The French Revolutionary calendar was created by the Jacobins after the 1789 revolution, to represent a more secular and nature-based view of the annual cycle, and to install a 10-day week in a rationalization measure similar to the metric system. The French government officially abandoned this calendar at the end of 1805.

The Maya of Central America used three separate, overlapping calendar systems, the *long count*, the *tzolkin*, and the *haab*. Emacs knows about all three of these calendars. Experts dispute the exact correlation between the Mayan calendar and our calendar; Emacs uses the Goodman-Martinez-Thompson correlation in its calculations.

The Copts use a calendar based on the ancient Egyptian solar calendar. Their calendar consists of twelve 30-day months followed by an extra five-day period. Once every fourth

year they add a leap day to this extra period to make it six days. The Ethiopic calendar is identical in structure, but has different year numbers and month names.

The Persians use a solar calendar based on a design of Omar Khayyam. Their calendar consists of twelve months of which the first six have 31 days, the next five have 30 days, and the last has 29 in ordinary years and 30 in leap years. Leap years occur in a complicated pattern every four or five years. The calendar implemented here is the arithmetical Persian calendar championed by Birashk, based on a 2,820-year cycle. It differs from the astronomical Persian calendar, which is based on astronomical events. As of this writing the first future discrepancy is projected to occur on March 20, 2025. It is currently not clear what the official calendar of Iran will be at that time.

The Chinese calendar is a complicated system of lunar months arranged into solar years. The years go in cycles of sixty, each year containing either twelve months in an ordinary year or thirteen months in a leap year; each month has either 29 or 30 days. Years, ordinary months, and days are named by combining one of ten "celestial stems" with one of twelve "terrestrial branches" for a total of sixty names that are repeated in a cycle of sixty.

The Bahá'í calendar system is based on a solar cycle of 19 months with 19 days each. The four remaining "intercalary" days are placed between the 18th and 19th months.

### 28.9.2 Converting To Other Calendars

The following commands describe the selected date (the date at point) in various other calendar systems:

`Mouse-3 Other calendars`

`p o` Display the selected date in various other calendars. (`calendar-print-other-dates`).

`p c` Display ISO commercial calendar equivalent for selected day (`calendar-iso-print-date`).

`p j` Display Julian date for selected day (`calendar-julian-print-date`).

`p a` Display astronomical (Julian) day number for selected day (`calendar-astro-print-day-number`).

`p h` Display Hebrew date for selected day (`calendar-hebrew-print-date`).

`p i` Display Islamic date for selected day (`calendar-islamic-print-date`).

`p f` Display French Revolutionary date for selected day (`calendar-french-print-date`).

`p b` Display Bahá'í date for selected day (`calendar-bahai-print-date`).

`p C` Display Chinese date for selected day (`calendar-chinese-print-date`).

`p k` Display Coptic date for selected day (`calendar-coptic-print-date`).

`p e` Display Ethiopic date for selected day (`calendar-ethiopic-print-date`).

`p p` Display Persian date for selected day (`calendar-persian-print-date`).

`p m` Display Mayan date for selected day (`calendar-mayan-print-date`).

Otherwise, move point to the date you want to convert, then type the appropriate command starting with `p` from the table above. The prefix `p` is a mnemonic for "print", since Emacs "prints" the equivalent date in the echo area. `p o` displays the date in all forms known to Emacs. You can also use `Mouse-3` and then choose `Other calendars` from the menu that appears. This displays the equivalent forms of the date in all the calendars Emacs understands, in the form of a menu. (Choosing an alternative from this menu doesn't actually do anything—the menu is used only for display.)

### 28.9.3 Converting From Other Calendars

You can use the other supported calendars to specify a date to move to. This section describes the commands for doing this using calendars other than Mayan; for the Mayan calendar, see the following section.

`g c` Move to a date specified in the ISO commercial calendar (`calendar-iso-goto-date`).

`g w` Move to a week specified in the ISO commercial calendar (`calendar-iso-goto-week`).

`g j` Move to a date specified in the Julian calendar (`calendar-julian-goto-date`).

`g a` Move to a date specified with an astronomical (Julian) day number (`calendar-astro-goto-day-number`).

`g b` Move to a date specified in the Bahá'í calendar (`calendar-bahai-goto-date`).

`g h` Move to a date specified in the Hebrew calendar (`calendar-hebrew-goto-date`).

`g i` Move to a date specified in the Islamic calendar (`calendar-islamic-goto-date`).

`g f` Move to a date specified in the French Revolutionary calendar (`calendar-french-goto-date`).

`g C` Move to a date specified in the Chinese calendar (`calendar-chinese-goto-date`).

`g p` Move to a date specified in the Persian calendar (`calendar-persian-goto-date`).

`g k` Move to a date specified in the Coptic calendar (`calendar-coptic-goto-date`).

`g e` Move to a date specified in the Ethiopic calendar (`calendar-ethiopic-goto-date`).

These commands ask you for a date on the other calendar, move point to the Gregorian calendar date equivalent to that date, and display the other calendar's date in the echo area. Emacs uses strict completion (see Section 5.4.3 [Completion Exit], page 72) whenever it asks you to type a month name, so you don't have to worry about the spelling of Hebrew, Islamic, or French names.

One common issue concerning the Hebrew calendar is the computation of the anniversary of a date of death, called a "yahrzeit". The Emacs calendar includes a facility for such calculations. If you are in the calendar, the command `M-x calendar-hebrew-list-yahrzeits`

asks you for a range of years and then displays a list of the yahrzeit dates for those years for the date given by point. If you are not in the calendar, this command first asks you for the date of death and the range of years, and then displays the list of yahrzeit dates.

### 28.9.4 Converting from the Mayan Calendar

Here are the commands to select dates based on the Mayan calendar:

`g m l` Move to a date specified by the long count calendar (`calendar-mayan-goto-long-count-date`).

`g m n t` Move to the next occurrence of a place in the tzolkin calendar (`calendar-mayan-next-tzolkin-date`).

`g m p t` Move to the previous occurrence of a place in the tzolkin calendar (`calendar-mayan-previous-tzolkin-date`).

`g m n h` Move to the next occurrence of a place in the haab calendar (`calendar-mayan-next-haab-date`).

`g m p h` Move to the previous occurrence of a place in the haab calendar (`calendar-mayan-previous-haab-date`).

`g m n c` Move to the next occurrence of a place in the calendar round (`calendar-mayan-next-calendar-round-date`).

`g m p c` Move to the previous occurrence of a place in the calendar round (`calendar-mayan-previous-calendar-round-date`).

To understand these commands, you need to understand the Mayan calendars. The *long count* is a counting of days with these units:

1 kin = 1 day  1 uinal = 20 kin  1 tun = 18 uinal
1 katun = 20 tun  1 baktun = 20 katun

Thus, the long count date 12.16.11.16.6 means 12 baktun, 16 katun, 11 tun, 16 uinal, and 6 kin. The Emacs calendar can handle Mayan long count dates as early as 7.17.18.13.3, but no earlier. When you use the `g m l` command, type the Mayan long count date with the baktun, katun, tun, uinal, and kin separated by periods.

The Mayan tzolkin calendar is a cycle of 260 days formed by a pair of independent cycles of 13 and 20 days. Since this cycle repeats endlessly, Emacs provides commands to move backward and forward to the previous or next point in the cycle. Type `g m p t` to go to the previous tzolkin date; Emacs asks you for a tzolkin date and moves point to the previous occurrence of that date. Similarly, type `g m n t` to go to the next occurrence of a tzolkin date.

The Mayan haab calendar is a cycle of 365 days arranged as 18 months of 20 days each, followed by a 5-day monthless period. Like the tzolkin cycle, this cycle repeats endlessly, and there are commands to move backward and forward to the previous or next point in the cycle. Type `g m p h` to go to the previous haab date; Emacs asks you for a haab date and moves point to the previous occurrence of that date. Similarly, type `g m n h` to go to the next occurrence of a haab date.

The Maya also used the combination of the tzolkin date and the haab date. This combination is a cycle of about 52 years called a *calendar round*. If you type `g m p c`, Emacs asks

you for both a haab and a tzolkin date and then moves point to the previous occurrence of that combination. Use `g m n c` to move point to the next occurrence of a combination. These commands signal an error if the haab/tzolkin date combination you have typed is impossible.

Emacs uses strict completion (see Section 5.4.3 [Completion Exit], page 72) whenever it asks you to type a Mayan name, so you don't have to worry about spelling.

## 28.10 The Diary

The Emacs diary keeps track of appointments or other events on a daily basis, in conjunction with the calendar. To use the diary feature, you must first create a *diary file* containing a list of events and their dates. Then Emacs can automatically pick out and display the events for today, for the immediate future, or for any specified date.

The name of the diary file is specified by the variable `diary-file`; '`~/diary`' is the default. Here's an example showing what that file looks like:

```
12/22/2012  Twentieth wedding anniversary!!
&1/1.       Happy New Year!
10/22       Ruth's birthday.
* 21, *:    Payday
Tuesday--weekly meeting with grad students at 10am
         Supowit, Shen, Bitner, and Kapoor to attend.
1/13/89     Friday the thirteenth!!
&thu 4pm    squash game with Lloyd.
mar 16      Dad's birthday
April 15, 2013 Income tax due.
&* 15       time cards due.
```

This format is essentially the same as the one used by the separate `calendar` utility that is present on some Unix systems. This example uses extra spaces to align the event descriptions of most of the entries. Such formatting is purely a matter of taste.

Although you probably will start by creating a diary manually, Emacs provides a number of commands to let you view, add, and change diary entries.

### 28.10.1 Displaying the Diary

Once you have created a diary file, you can use the calendar to view it. You can also view today's events outside of Calendar mode. In the following, key bindings refer to the Calendar buffer.

`Mouse-3 Diary`

`d` Display all diary entries for the selected date (`diary-view-entries`).

`s` Display the entire diary file (`diary-show-all-entries`).

`m` Mark all visible dates that have diary entries (`diary-mark-entries`).

`u` Unmark the calendar window (`calendar-unmark`).

`M-x diary-print-entries`
Print hard copy of the diary display as it appears.

`M-x diary` Display all diary entries for today's date.

`M-x diary-mail-entries`
: Mail yourself email reminders about upcoming diary entries.

Displaying the diary entries with `d` shows in a separate window the diary entries for the selected date in the calendar. The mode line of the new window shows the date of the diary entries. Holidays are shown either in the buffer or in the mode line, depending on the display method you choose (see Section "Diary Display" in *Specialized Emacs Features*). If you specify a numeric argument with `d`, it shows all the diary entries for that many successive days. Thus, `2 d` displays all the entries for the selected date and for the following day.

Another way to display the diary entries for a date is to click `Mouse-3` on the date, and then choose `Diary entries` from the menu that appears. If the variable `calendar-view-diary-initially-flag` is non-`nil`, creating the calendar lists the diary entries for the current date (provided the current date is visible).

To get a broader view of which days are mentioned in the diary, use the `m` command. This marks the dates that have diary entries in a different face. See Section "Calendar Customizing" in *Specialized Emacs Features*.

This command applies both to the months that are currently visible and to those that subsequently become visible after scrolling. To turn marking off and erase the current marks, type `u`, which also turns off holiday marks (see Section 28.6 [Holidays], page 608). If the variable `calendar-mark-diary-entries-flag` is non-`nil`, creating or updating the calendar marks diary dates automatically.

To see the full diary file, rather than just some of the entries, use the `s` command.

The command `M-x diary` displays the diary entries for the current date, independently of the calendar display, and optionally for the next few days as well; the variable `diary-number-of-entries` specifies how many days to include. See Section "Diary Customizing" in *Specialized Emacs Features*.

If you put `(diary)` in your '`.emacs`' file, this automatically displays a window with the day's diary entries when you start Emacs.

Some people like to receive email notifications of events in their diary. To send such mail to yourself, use the command `M-x diary-mail-entries`. A prefix argument specifies how many days (starting with today) to check; otherwise, the variable `diary-mail-days` says how many days.

### 28.10.2 The Diary File

Your *diary file* is a file that records events associated with particular dates. The name of the diary file is specified by the variable `diary-file`; '`~/diary`' is the default. The `calendar` utility program supports a subset of the format allowed by the Emacs diary facilities, so you can use that utility to view the diary file, with reasonable results aside from the entries it cannot understand.

Each entry in the diary file describes one event and consists of one or more lines. An entry always begins with a date specification at the left margin. The rest of the entry is simply text to describe the event. If the entry has more than one line, then the lines after the first must begin with whitespace to indicate they continue a previous entry. Lines that do not begin with valid dates and do not continue a preceding entry are ignored.

You can also use a format where the first line of a diary entry consists only of the date or day name (with no following blanks or punctuation). For example:

```
02/11/2012
      Bill B. visits Princeton today
      2pm Cognitive Studies Committee meeting
      2:30-5:30 Liz at Lawrenceville
      4:00pm Dentist appt
      7:30pm Dinner at George's
      8:00-10:00pm concert
```

This entry will have a different appearance if you use the simple diary display (see Section "Diary Display" in *Specialized Emacs Features*). The simple diary display omits the date line at the beginning; only the continuation lines appear. This style of entry looks neater when you display just a single day's entries, but can cause confusion if you ask for more than one day's entries.

You can inhibit the marking of certain diary entries in the calendar window; to do this, insert the string that `diary-nonmarking-symbol` specifies (default '`&`') at the beginning of the entry, before the date. This has no effect on display of the entry in the diary window; it only affects marks on dates in the calendar window. Nonmarking entries are especially useful for generic entries that would otherwise mark many different dates.

### 28.10.3 Date Formats

Here are some sample diary entries, illustrating different ways of formatting a date. The examples all show dates in American order (month, day, year), but Calendar mode supports European order (day, month, year) and ISO order (year, month, day) as options.

```
4/20/12  Switch-over to new tabulation system
apr. 25  Start tabulating annual results
4/30  Results for April are due
*/25  Monthly cycle finishes
Friday  Don't leave without backing up files
```

The first entry appears only once, on April 20, 2012. The second and third appear every year on the specified dates, and the fourth uses a wildcard (asterisk) for the month, so it appears on the 25th of every month. The final entry appears every week on Friday.

You can use just numbers to express a date, as in '*`month/day`*' or '*`month/day/year`*'. This must be followed by a nondigit. In the date itself, *month* and *day* are numbers of one or two digits. The optional *year* is also a number, and may be abbreviated to the last two digits; that is, you can use '`11/12/2012`' or '`11/12/12`'.

Dates can also have the form '*`monthname day`*' or '*`monthname day, year`*', where the month's name can be spelled in full or abbreviated (with or without a period). The preferred abbreviations for month and day names can be set using the variables `calendar-abbrev-length`, `calendar-month-abbrev-array`, and `calendar-day-abbrev-array`. The default is to use the first three letters of a name as its abbreviation. Case is not significant.

A date may be *generic*; that is, partially unspecified. Then the entry applies to all dates that match the specification. If the date does not contain a year, it is generic and applies to any year. Alternatively, *month*, *day*, or *year* can be '`*`'; this matches any month, day, or year, respectively. Thus, a diary entry '`3/*/*`' matches any day in March of any year; so does '`march *`'.

If you prefer the European style of writing dates (in which the day comes before the month), or the ISO style (in which the order is year, month, day), type `M-x calendar-set-date-style` while in the calendar, or customize the variable `calendar-date-style`. This affects how diary dates are interpreted, date display, and the order in which some commands expect their arguments to be given.

You can use the name of a day of the week as a generic date which applies to any date falling on that day of the week. You can abbreviate the day of the week as described above, or spell it in full; case is not significant.

### 28.10.4 Commands to Add to the Diary

While in the calendar, there are several commands to create diary entries. The basic commands are listed here; more sophisticated commands are in the next section (see Section 28.10.5 [Special Diary Entries], page 618). Entries can also be based on non-Gregorian calendars. See Section "Non-Gregorian Diary" in *Specialized Emacs Features*.

`i d` Add a diary entry for the selected date (`diary-insert-entry`).

`i w` Add a diary entry for the selected day of the week (`diary-insert-weekly-entry`).

`i m` Add a diary entry for the selected day of the month (`diary-insert-monthly-entry`).

`i y` Add a diary entry for the selected day of the year (`diary-insert-yearly-entry`).

You can make a diary entry for a specific date by selecting that date in the calendar window and typing the `i d` command. This command displays the end of your diary file in another window and inserts the date; you can then type the rest of the diary entry.

If you want to make a diary entry that applies to a specific day of the week, select that day of the week (any occurrence will do) and type `i w`. This inserts the day-of-week as a generic date; you can then type the rest of the diary entry. You can make a monthly diary entry in the same fashion: select the day of the month, use the `i m` command, and type the rest of the entry. Similarly, you can insert a yearly diary entry with the `i y` command.

All of the above commands make marking diary entries by default. To make a nonmarking diary entry, give a prefix argument to the command. For example, `C-u i w` makes a nonmarking weekly diary entry.

When you modify the diary file, be sure to save the file before exiting Emacs. Saving the diary file after using any of the above insertion commands will automatically update the diary marks in the calendar window, if appropriate. You can use the command `calendar-redraw` to force an update at any time.

### 28.10.5 Special Diary Entries

In addition to entries based on calendar dates, the diary file can contain *sexp entries* for regular events such as anniversaries. These entries are based on Lisp expressions (sexps) that Emacs evaluates as it scans the diary file. Instead of a date, a sexp entry contains '`%%`' followed by a Lisp expression which must begin and end with parentheses. The Lisp expression determines which dates the entry applies to.

Calendar mode provides commands to insert certain commonly used sexp entries:

`i a` Add an anniversary diary entry for the selected date (`diary-insert-anniversary-entry`).

`i b` Add a block diary entry for the current region (`diary-insert-block-entry`).

`i c` Add a cyclic diary entry starting at the date (`diary-insert-cyclic-entry`).

If you want to make a diary entry that applies to the anniversary of a specific date, move point to that date and use the `i a` command. This displays the end of your diary file in another window and inserts the anniversary description; you can then type the rest of the diary entry. The entry looks like this:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday
```

This entry applies to October 31 in any year after 1948; '`10 31 1948`' specifies the date. (If you are using the European or ISO calendar style, the input order of month, day and year is different.) The reason this expression requires a beginning year is that advanced diary functions can use it to calculate the number of elapsed years.

A *block* diary entry applies to a specified range of consecutive dates. Here is a block diary entry that applies to all dates from June 24, 2012 through July 10, 2012:

```
%%(diary-block 6 24 2012 7 10 2012) Vacation
```

The '`6 24 2012`' indicates the starting date and the '`7 10 2012`' indicates the stopping date. (Again, if you are using the European or ISO calendar style, the input order of month, day and year is different.)

To insert a block entry, place point and the mark on the two dates that begin and end the range, and type `i b`. This command displays the end of your diary file in another window and inserts the block description; you can then type the diary entry.

*Cyclic* diary entries repeat after a fixed interval of days. To create one, select the starting date and use the `i c` command. The command prompts for the length of interval, then inserts the entry, which looks like this:

```
%%(diary-cyclic 50 3 1 2012) Renew medication
```

This entry applies to March 1, 2012 and every 50th day following; '`3 1 2012`' specifies the starting date. (If you are using the European or ISO calendar style, the input order of month, day and year is different.)

All three of these commands make marking diary entries. To insert a nonmarking entry, give a prefix argument to the command. For example, `C-u i a` makes a nonmarking anniversary diary entry.

Marking sexp diary entries in the calendar can be time-consuming, since every date visible in the calendar window must be individually checked. So it's a good idea to make sexp diary entries nonmarking (with '`&`') when possible.

Another sophisticated kind of sexp entry, a *floating* diary entry, specifies a regularly occurring event by offsets specified in days, weeks, and months. It is comparable to a crontab entry interpreted by the `cron` utility. Here is a nonmarking, floating diary entry that applies to the fourth Thursday in November:

```
&%%(diary-float 11 4 4) American Thanksgiving
```

The 11 specifies November (the eleventh month), the 4 specifies Thursday (the fourth day of the week, where Sunday is numbered zero), and the second 4 specifies the fourth Thursday

(1 would mean "first", 2 would mean "second", −2 would mean "second-to-last", and so on). The month can be a single month or a list of months. Thus you could change the 11 above to '`'(1 2 3)`' and have the entry apply to the last Thursday of January, February, and March. If the month is `t`, the entry applies to all months of the year.

Each of the standard sexp diary entries takes an optional parameter specifying the name of a face or a single-character string to use when marking the entry in the calendar. Most generally, sexp diary entries can perform arbitrary computations to determine when they apply. See Section "Sexp Diary Entries" in *Specialized Emacs Features*.

## 28.11 Appointments

If you have a diary entry for an appointment, and that diary entry begins with a recognizable time of day, Emacs can warn you in advance that an appointment is pending. Emacs alerts you to the appointment by displaying a message in your chosen format, as specified by the variable `appt-display-format`. If the value of `appt-audible` is non-`nil`, the warning includes an audible reminder. In addition, if `appt-display-mode-line` is non-`nil`, Emacs displays the number of minutes to the appointment on the mode line.

If `appt-display-format` has the value `window`, then the variable `appt-display-duration` controls how long the reminder window is visible for; and the variables `appt-disp-window-function` and `appt-delete-window-function` give the names of functions used to create and destroy the window, respectively.

To enable appointment notification, type `M-x appt-activate`. With a positive argument, it enables notification; with a negative argument, it disables notification; with no argument, it toggles. Enabling notification also sets up an appointment list for today from the diary file, giving all diary entries found with recognizable times of day, and reminds you just before each of them.

For example, suppose the diary file contains these lines:

```
Monday
  9:30am Coffee break
 12:00pm Lunch
```

Then on Mondays, you will be reminded at around 9:20am about your coffee break and at around 11:50am about lunch. The variable `appt-message-warning-time` specifies how many minutes (default 12) in advance to warn you. This is a default warning time. Each appointment can specify a different warning time by adding a piece matching `appt-warning-time-regexp` (see that variable's documentation for details).

You can write times in am/pm style (with '`12:00am`' standing for midnight and '`12:00pm`' standing for noon), or 24-hour European/military style. You need not be consistent; your diary file can have a mixture of the two styles. Times must be at the beginning of diary entries if they are to be recognized.

Emacs updates the appointments list from the diary file automatically just after midnight. You can force an update at any time by re-enabling appointment notification. Both these actions also display the day's diary buffer, unless you set `appt-display-diary` to `nil`. The appointments list is also updated whenever the diary file (or a file it includes; see Section "Fancy Diary Display" in *Specialized Emacs Features*) is saved.

You can also use the appointment notification facility like an alarm clock. The command `M-x appt-add` adds entries to the appointment list without affecting your diary file. You delete entries from the appointment list with `M-x appt-delete`.

## 28.12 Importing and Exporting Diary Entries

You can transfer diary entries between Emacs diary files and a variety of other formats.

You can import diary entries from Outlook-generated appointment messages. While viewing such a message in Rmail or Gnus, do `M-x diary-from-outlook` to import the entry. You can make this command recognize additional appointment message formats by customizing the variable `diary-outlook-formats`.

The icalendar package allows you to transfer data between your Emacs diary file and iCalendar files, which are defined in "RFC 2445—Internet Calendaring and Scheduling Core Object Specification (iCalendar)" (as well as the earlier vCalendar format).

The command `icalendar-import-buffer` extracts iCalendar data from the current buffer and adds it to your diary file. This function is also suitable for automatic extraction of iCalendar data; for example with the Rmail mail client one could use:

```
(add-hook 'rmail-show-message-hook 'icalendar-import-buffer)
```

The command `icalendar-import-file` imports an iCalendar file and adds the results to an Emacs diary file. For example:

```
(icalendar-import-file "/here/is/calendar.ics"
                       "/there/goes/ical-diary")
```

You can use an `#include` directive to add the import file contents to the main diary file, if these are different files. See Section "Fancy Diary Display" in *Specialized Emacs Features*.

Use `icalendar-export-file` to interactively export an entire Emacs diary file to iCalendar format. To export only a part of a diary file, mark the relevant area, and call `icalendar-export-region`. In both cases, Emacs appends the result to the target file.

## 28.13 Daylight Saving Time

Emacs understands the difference between standard time and daylight saving time—the times given for sunrise, sunset, solstices, equinoxes, and the phases of the moon take that into account. The rules for daylight saving time vary from place to place and have also varied historically from year to year. To do the job properly, Emacs needs to know which rules to use.

Some operating systems keep track of the rules that apply to the place where you are; on these systems, Emacs gets the information it needs from the system automatically. If some or all of this information is missing, Emacs fills in the gaps with the rules currently used in Cambridge, Massachusetts. If the resulting rules are not what you want, you can tell Emacs the rules to use by setting certain variables: `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends`.

These values should be Lisp expressions that refer to the variable `year`, and evaluate to the Gregorian date on which daylight saving time starts or (respectively) ends, in the form of a list `(month day year)`. The values should be `nil` if your area does not use daylight saving time.

Emacs uses these expressions to determine the starting date of daylight saving time for the holiday list and for correcting times of day in the solar and lunar calculations.

The values for Cambridge, Massachusetts are as follows:

```
(calendar-nth-named-day 2 0 3 year)
(calendar-nth-named-day 1 0 11 year)
```

That is, the second 0th day (Sunday) of the third month (March) in the year specified by `year`, and the first Sunday of the eleventh month (November) of that year. If daylight saving time were changed to start on October 1, you would set `calendar-daylight-savings-starts` to this:

```
(list 10 1 year)
```

If there is no daylight saving time at your location, or if you want all times in standard time, set `calendar-daylight-savings-starts` and `calendar-daylight-savings-ends` to `nil`.

The variable `calendar-daylight-time-offset` specifies the difference between daylight saving time and standard time, measured in minutes. The value for Cambridge, Massachusetts is 60.

Finally, the two variables `calendar-daylight-savings-starts-time` and `calendar-daylight-savings-ends-time` specify the number of minutes after midnight local time when the transition to and from daylight saving time should occur. For Cambridge, Massachusetts both variables' values are 120.

## 28.14 Summing Time Intervals

The timeclock package adds up time intervals, so you can (for instance) keep track of how much time you spend working on particular projects.

Use the `M-x timeclock-in` command when you start working on a project, and `M-x timeclock-out` command when you're done. Each time you do this, it adds one time interval to the record of the project. You can change to working on a different project with `M-x timeclock-change`.

Once you've collected data from a number of time intervals, you can use `M-x timeclock-workday-remaining` to see how much time is left to work today (assuming a typical average of 8 hours a day), and `M-x timeclock-when-to-leave` which will calculate when you're "done".

If you want Emacs to display the amount of time "left" of your workday in the mode line, either customize the `timeclock-modeline-display` variable and set its value to `t`, or invoke the `M-x timeclock-modeline-display` command.

Terminating the current Emacs session might or might not mean that you have stopped working on the project and, by default, Emacs asks you. You can, however, customize the value of the variable `timeclock-ask-before-exiting` to `nil` to avoid the question; then, only an explicit `M-x timeclock-out` or `M-x timeclock-change` will tell Emacs that the current interval is over.

The timeclock functions work by accumulating the data in a file called '`.timelog`' in your home directory. You can specify a different name for this file by customizing the variable `timeclock-file`. If you edit the timeclock file manually, or if you change

the value of any of timeclock's customizable variables, you should run the command `M-x timeclock-reread-log` to update the data in Emacs from the file.

# 29 Sending Mail

To send an email message from Emacs, type `C-x m`. This switches to a buffer named '`*unsent mail*`', where you can edit the text and headers of the message. When done, type `C-c C-s` or `C-c C-c` to send it.

`C-x m` Begin composing mail (`compose-mail`).

`C-x 4 m` Likewise, in another window (`compose-mail-other-window`).

`C-x 5 m` Likewise, but in a new frame (`compose-mail-other-frame`).

`C-c C-s` In the mail buffer, send the message (`message-send`).

`C-c C-c` In the mail buffer, send the message and bury the buffer (`message-send-and-exit`).

The mail buffer is an ordinary Emacs buffer, so you can switch to other buffers while composing the mail. If you want to send another mail before finishing the current one, type `C-x m` again to open a new mail buffer whose name has a different numeric suffix (see ⟨undefined⟩ [Misc Buffer], page ⟨undefined⟩). If you invoke the command with a prefix argument, `C-u C-x m`, Emacs switches back to the last mail buffer, and asks if you want to erase the message in that buffer; if you answer no, this lets you pick up editing the message where you left off.

The command `C-x 4 m` (`compose-mail-other-window`) does the same as `C-x m`, except it displays the mail buffer in a different window. The command `C-x 5 m` (`compose-mail-other-frame`) does it in a new frame.

When you type `C-c C-c` or `C-c C-s` to send the mail, Emacs may ask you how it should deliver the mail—either directly via SMTP, or using some other method. See Section 29.4.1 [Mail Sending], page 627, for details.

## 29.1 The Format of the Mail Buffer

Here is an example of the contents of a mail buffer:

```
To: subotai@example.org
CC: mongol.soldier@example.net, rms@gnu.org
Subject: Re: What is best in life?
From: conan@example.org
--text follows this line--
To crush your enemies, see them driven before you, and to
hear the lamentation of their women.
```

At the top of the mail buffer is a set of *header fields*, which are used for specifying information about the email's recipient(s), subject, and so on. The above buffer contains header fields for '`To`', '`Cc`', '`Subject`', and '`From`'. Some header fields are automatically pre-initialized in the mail buffer, when appropriate.

The line that says '`--text follows this line--`' separates the header fields from the *body* (or *text*) of the message. Everything above that line is treated as part of the headers; everything below it is treated as the body. The delimiter line itself does not appear in the message actually sent.

You can insert and edit header fields using ordinary editing commands. See Section 29.4.2 [Header Editing], page 628, for commands specific to editing header fields. Certain headers, such as '`Date`' and '`Message-Id`', are normally omitted from the mail buffer and are created automatically when the message is sent.

## 29.2 Mail Header Fields

A header field in the mail buffer starts with a field name at the beginning of a line, terminated by a colon. Upper and lower case are equivalent in field names. After the colon and optional whitespace comes the contents of the field.

You can use any name you like for a header field, but normally people use only standard field names with accepted meanings.

The '`From`' header field identifies the person sending the email (i.e., you). This should be a valid mailing address, as replies are normally sent there. The default contents of this header field are computed from the variables `user-full-name` (which specifies your full name) and `user-mail-address` (your email address). On some operating systems, Emacs initializes these two variables using environment variables (see Section E.4.1 [General Variables], page 758). If this information is unavailable or wrong, you should customize the variables yourself (see Section 33.1 [Easy Customization], page 686).

The value of the variable `mail-from-style` specifies how to format the contents of the '`From`' field:

`nil` Use just the address, as in '`king@grassland.com`'.

`parens` Use both address and full name, as in:
'`king@grassland.com (Elvis Parsley)`'.

`angles` Use both address and full name, as in:
'`Elvis Parsley <king@grassland.com>`'.

any other value
Use `angles` normally. But if the address must be "quoted" to remain syntactically valid under the `angles` format but not under the `parens` format, use `parens` instead. This is the default.

Apart from '`From`', here is a table of commonly-used fields:

'`To`' The mailing address(es) to which the message is addressed. To list more than one address, use commas to separate them.

'`Subject`' The subject of the message.

'`CC`' Additional mailing address(es) to send the message to. This is like '`To`', except that these readers should not regard the message as directed at them.

'`BCC`' Additional mailing address(es) to send the message to, which should not appear in the header of the message actually sent. "BCC" stands for *blind carbon copies*.

'`FCC`' The name of a file, to which a copy of the sent message should be appended. Emacs writes the message in mbox format, unless the file is in Babyl format (used by Rmail before Emacs 23), in which case Emacs writes in Babyl format.

If an Rmail buffer is visiting the file, Emacs updates it accordingly. To specify more than one file, use several '`FCC`' fields, with one file name in each field.

'`Reply-to`'
: An address to which replies should be sent, instead of '`From`'. This is used if, for some reason, your '`From`' address cannot receive replies.

'`Mail-reply-to`'
: This field takes precedence over '`Reply-to`'. It is used because some mailing lists set the '`Reply-to`' field for their own purposes (a somewhat controversial practice).

'`Mail-followup-to`'
: One of more address(es) to use as default recipient(s) for follow-up messages. This is typically used when you reply to a message from a mailing list that you are subscribed to, and want replies to go to the list without sending an extra copy to you.

'`In-reply-to`'
: An identifier for the message you are replying to. Most mail readers use this information to group related messages together. Normally, this header is filled in automatically when you reply to a message in any mail program built into Emacs.

'`References`'
: Identifiers for previous related messages. Like '`In-reply-to`', this is normally filled in automatically for you.

The '`To`', '`CC`', and '`BCC`' fields can appear any number of times, and each such header field can contain multiple addresses, separated by commas. This way, you can specify any number of places to send the message. These fields can also have continuation lines: one or more lines starting with whitespace, following the starting line of the field, are considered part of the field. Here's an example of a '`To`' field with a continuation line:

```
To: foo@example.net, this@example.net,
  bob@example.com
```

You can direct Emacs to insert certain default headers into the mail buffer by setting the variable `mail-default-headers` to a string. Then `C-x m` inserts this string into the message headers. For example, here is how to add a '`Reply-to`' and '`FCC`' header to each message:

```
(setq mail-default-headers
      "Reply-to: foo@example.com\nFCC: ~/Mail/sent")
```

If the default header fields are not appropriate for a particular message, edit them as necessary before sending the message.

## 29.3 Mail Aliases

You can define *mail aliases*, which are short mnemonic names that stand for one or more mailing addresses. By default, mail aliases are defined in the file '`~/.mailrc`'. You can specify a different file name to use, by setting the variable `mail-personal-alias-file`.

To define an alias in '`.mailrc`', write a line like this:

```
alias nick fulladdresses
```

This means that *nick* should expand into *fulladdresses*, where *fulladdresses* can be either a single address, or multiple addresses separated with spaces. For instance, to make `maingnu` stand for `gnu@gnu.org` plus a local address of your own, put in this line:

```
alias maingnu gnu@gnu.org local-gnu
```

If an address contains a space, quote the whole address with a pair of double quotes, like this:

```
alias jsmith "John Q. Smith <none@example.com>"
```

Note that you need not include double quotes around individual parts of the address, such as the person's full name. Emacs puts them in if they are needed. For instance, it inserts the above address as '`"John Q. Smith" <none@example.com>`'.

Emacs also recognizes "include" commands in '`.mailrc`'. They look like this:

```
source filename
```

The '`.mailrc`' file is not unique to Emacs; many other mail-reading programs use it for mail aliases, and it can contain various other commands. However, Emacs ignores everything except alias definitions and include commands.

Mail aliases expand as abbrevs—that is to say, as soon as you type a word-separator character after an alias (see Chapter 26 [Abbrevs], page 581). This expansion takes place only within the '`To`', '`From`', '`CC`', '`BCC`', and '`Reply-to`' header fields (plus their '`Resent-`' variants); it does not take place in other header fields, such as '`Subject`'.

You can also insert an aliased address directly, using the command `M-x mail-abbrev-insert-alias`. This reads an alias name, with completion, and inserts its definition at point.

## 29.4 Mail Commands

The default major mode for the '`*mail*`' buffer is called Message mode. It behaves like Text mode in many ways, but provides several additional commands on the `C-c` prefix, which make editing a message more convenient.

In this section, we will describe some of the most commonly-used commands available in Message mode.

### 29.4.1 Mail Sending

`C-c C-c` Send the message, and bury the mail buffer (`message-send-and-exit`).

`C-c C-s` Send the message, and leave the mail buffer selected (`message-send`).

The usual command to send a message is `C-c C-c` (`mail-send-and-exit`). This sends the message and then "buries" the mail buffer, putting it at the lowest priority for reselection. If you want it to kill the mail buffer instead, change the variable `message-kill-buffer-on-exit` to `t`.

The command `C-c C-s` (`message-send`) sends the message and leaves the buffer selected. Use this command if you want to modify the message (perhaps with new recipients) and send it again.

Sending a message runs the hook `message-send-hook`. It also marks the mail buffer as unmodified, except if the mail buffer is also a file-visiting buffer (in that case, only saving the file does that, and you don't get a warning if you try to send the same message twice).

The variable `send-mail-function` controls how the message is delivered. Its value should be one of the following functions:

`sendmail-query-once`
: Query for a delivery method (one of the other entries in this list), and use that method for this message; then save the method to `send-mail-function`, so that it is used for future deliveries. This is the default, unless you have already set the variables for sending mail via `smtpmail-send-it` (see below).

`smtpmail-send-it`
: Send mail using the through an external mail host, such as your Internet service provider's outgoing SMTP mail server. If you have not told Emacs how to contact the SMTP server, it prompts for this information, which is saved in the `smtpmail-smtp-server` variable and the file '`~/.authinfo`'. See Section "Emacs SMTP Library" in *Sending mail via SMTP*.

`sendmail-send-it`
: Send mail using the system's default `sendmail` program, or equivalent. This requires the system to be set up for delivering mail directly via SMTP.

`mailclient-send-it`
: Pass the mail buffer on to the system's designated mail client. See the commentary section in the file '`mailclient.el`' for details.

`feedmail-send-it`
: This is similar to `sendmail-send-it`, but allows you to queue messages for later sending. See the commentary section in the file '`feedmail.el`' for details.

When you send a message containing non-ASCII characters, they need to be encoded with a coding system (see Section 19.6 [Coding Systems], page 381). Usually the coding system is specified automatically by your chosen language environment (see Section 19.3 [Language Environments], page 377). You can explicitly specify the coding system for outgoing mail by setting the variable `sendmail-coding-system` (see Section 19.7 [Recognize Coding], page 383). If the coding system thus determined does not handle the characters in a particular message, Emacs asks you to select the coding system to use, showing a list of possible coding systems.

### 29.4.2 Mail Header Editing

Message mode provides the following special commands to move to particular header fields and to complete addresses in headers.

`C-c C-f C-t`
: Move to the '`To`' header (`message-goto-to`).

`C-c C-f C-s`
: Move to the '`Subject`' header (`message-goto-subject`).

`C-c C-f C-c`
: Move to the '`CC`' header (`message-goto-cc`).

`C-c C-f C-b`
: Move to the 'BCC' header (`message-goto-bcc`).

`C-c C-f C-r`
: Move to the 'Reply-To' header (`message-goto-reply-to`).

`C-c C-f C-f`
: Move to the 'Mail-Followup-To' header field (`message-goto-followup-to`).

`C-c C-f C-w`
: Add a new 'FCC' header field, with file-name completion (`message-goto-fcc`).

`C-c C-b` Move to the start of the message body (`message-goto-body`).

TAB Complete a mailing address (`message-tab`).

The commands to move point to particular header fields are all based on the prefix `C-c C-f` ('`C-f`' is for "field"). If the field in question does not exist, the command creates one (the exception is `mail-fcc`, which creates a new field each time).

The command `C-c C-b` (`message-goto-body`) moves point to just after the header separator line—that is, to the beginning of the body.

While editing a header field that contains addresses, such as '`To:`', '`CC:`' and '`BCC:`', you can complete an address by typing TAB (`message-tab`). This attempts to insert the full name corresponding to the address based on a couple of methods, including EUDC, a library that recognizes a number of directory server protocols (see Section "EUDC" in *The Emacs Unified Directory Client*). Failing that, it attempts to expand the address as a mail alias (see Section 29.3 [Mail Aliases], page 626). If point is on a header field that does not take addresses, or if it is in the message body, then TAB just inserts a tab character.

### 29.4.3 Citing Mail

`C-c C-y` Yank the selected message from the mail reader, as a citation (`message-yank-original`).

`C-c C-q` Fill each paragraph cited from another message (`message-fill-yanked-message`).

You can use the command `C-c C-y` (`message-yank-original`) to *cite* a message that you are replying to. This inserts the text of that message into the mail buffer. This command works only if the mail buffer is invoked from a mail reader running in Emacs, such as Rmail.

By default, Emacs inserts the string '`>`' in front of each line of the cited text; this prefix string is specified by the variable `message-yank-prefix`. If you call `message-yank-original` with a prefix argument, the citation prefix is not inserted.

After using `C-c C-y`, you can type `C-c C-q` (`message-fill-yanked-message`) to fill the paragraphs of the cited message. One use of `C-c C-q` fills all such paragraphs, each one individually. To fill a single paragraph of the quoted message, use `M-q`. If filling does not automatically handle the type of citation prefix you use, try setting the fill prefix explicitly. See Section 22.11 [Filling], page 473.

You can customize mail citation through the hook `mail-citation-hook`. For example, you can use the Supercite package, which provides more flexible citation (see Section "Introduction" in *Supercite*).

### 29.4.4 Mail Miscellany

You can *attach* a file to an outgoing message by typing `C-c C-a` (`mml-attach-file`) in the mail buffer. Attaching is done using the Multipurpose Internet Mail Extensions (MIME) standard.

The `mml-attach-file` command prompts for the name of the file, and for the attachment's *content type*, *description*, and *disposition*. The content type is normally detected automatically; just type RET to accept the default. The description is a single line of text that the recipient will see next to the attachment; you may also choose to leave this empty. The disposition is either '`inline`' (the default), which means the recipient will see a link to the attachment within the message body, or '`attachment`', which means the link will be separate from the body.

The `mml-attach-file` command is specific to Message mode; in Mail mode use `mail-add-attachment` instead. It will prompt only for the name of the file, and will determine the content type and the disposition automatically. If you want to include some description of the attached file, type that in the message body.

The actual contents of the attached file are not inserted into the mail buffer. Instead, some placeholder text is inserted into the mail buffer, like this:

```
<#part type="text/plain" filename="~/foo.txt" disposition=inline>
<#/part>
```

When you type `C-c C-c` or `C-c C-s` to send the message, the attached file will be delivered with it.

While composing a message, you can do spelling correction on the message text by typing `M-x ispell-message`. If you have yanked an incoming message into the outgoing draft, this command skips what was yanked, but it checks the text that you yourself inserted (it looks for indentation or `mail-yank-prefix` to distinguish the cited lines from your input). See Section 13.4 [Spelling], page 219.

Turning on Message mode (which `C-x m` does automatically) runs the normal hooks `text-mode-hook` and `message-mode-hook`. Initializing a new outgoing message runs the normal hook `message-setup-hook`; you can use this hook if you want to make changes to the appearance of the mail buffer. See Section 33.2.2 [Hooks], page 696.

The main difference between these hooks is just when they are invoked. Whenever you type `C-x m`, `message-mode-hook` runs as soon as the mail buffer is created. Then the `message-setup` function inserts the default contents of the buffer. After these default contents are inserted, `message-setup-hook` runs.

If you use `C-x m` to continue an existing composition, `message-mode-hook` runs immediately after switching to the mail buffer. If the buffer is unmodified, or if you decide to erase it and start again, `message-setup-hook` runs after the default contents are inserted.

## 29.5 Mail Signature

You can add a standard piece of text—your *mail signature*—to the end of every message. This signature may contain information such as your telephone number or your physical location. The variable `mail-signature` determines how Emacs handles the mail signature.

The default value of `mail-signature` is `t`; this means to look for your mail signature in the file '`~/.signature`'. If this file exists, its contents are automatically inserted into the

end of the mail buffer. You can change the signature file via the variable `mail-signature-file`.

If you change `mail-signature` to a string, that specifies the text of the signature directly.

If you change `mail-signature` to `nil`, Emacs will not insert your mail signature automatically. You can insert your mail signature by typing `C-c C-w` (`message-insert-signature`) in the mail buffer. Emacs will look for your signature in the signature file.

By convention, a mail signature should be marked by a line whose contents are '`-- `'. If your signature lacks this prefix, it is added for you. The remainder of your signature should be no more than four lines.

## 29.6 Mail Amusements

`M-x spook` adds a line of randomly chosen keywords to an outgoing mail message. The keywords are chosen from a list of words that suggest you are discussing something subversive.

The idea behind this feature is the suspicion that the NSA[1] and other intelligence agencies snoop on all electronic mail messages that contain keywords suggesting they might find them interesting. (The agencies say that they don't, but that's what they *would* say.) The idea is that if lots of people add suspicious words to their messages, the agencies will get so busy with spurious input that they will have to give up reading it all. Whether or not this is true, it at least amuses some people.

You can use the `fortune` program to put a "fortune cookie" message into outgoing mail. To do this, add `fortune-to-signature` to `mail-setup-hook`:

```
(add-hook 'mail-setup-hook 'fortune-to-signature)
```

You will probably need to set the variable `fortune-file` before using this.

## 29.7 Mail-Composition Methods

In this chapter we have described the usual Emacs mode for editing and sending mail—Message mode. This is only one of several available modes. Prior to Emacs 23.2, the default mode was Mail mode, which is similar to Message mode in many respects but lacks features such as MIME support. Another available mode is MH-E (see Section "MH-E" in *The Emacs Interface to MH*).

You can choose any of these *mail user agents* as your preferred method for editing and sending mail. The commands `C-x m`, `C-x 4 m` and `C-x 5 m` use whichever agent you have specified; so do various other parts of Emacs that send mail, such as the bug reporter (see Section 34.3 [Bugs], page 722). To specify a mail user agent, customize the variable `mail-user-agent`. Currently, legitimate values include `message-user-agent` (Message mode) `sendmail-user-agent` (Mail mode), `gnus-user-agent`, and `mh-e-user-agent`.

If you select a different mail-composition method, the information in this chapter about the mail buffer and Message mode does not apply; the other methods use a different format of text in a different buffer, and their commands are different as well.

Similarly, to specify your preferred method for reading mail, customize the variable `read-mail-command`. The default is `rmail` (see Chapter 30 [Rmail], page 632).

[1] The US National Security Agency.

# 30 Reading Mail with Rmail

Rmail is an Emacs subsystem for reading and disposing of mail that you receive. Rmail stores mail messages in files called Rmail files. Reading the messages in an Rmail file is done in a special major mode, Rmail mode, which redefines most letters to run commands for managing mail.

## 30.1 Basic Concepts of Rmail

Using Rmail in the simplest fashion, you have one Rmail file '`~/RMAIL`' in which all of your mail is saved. It is called your *primary Rmail file*. The command `M-x rmail` reads your primary Rmail file, merges new mail in from your inboxes, displays the first message you haven't read yet, and lets you begin reading. The variable `rmail-file-name` specifies the name of the primary Rmail file.

Rmail displays only one message in the Rmail file at a time. The message that is shown is called the *current message*. Rmail mode's special commands can do such things as delete the current message, copy it into another file, send a reply, or move to another message. You can also create multiple Rmail files and use Rmail to move messages between them.

Within the Rmail file, messages are normally arranged sequentially in order of receipt; you can specify other ways to sort them (see Section 30.12 [Rmail Sorting], page 645). Messages are identified by consecutive integers which are their *message numbers*. The number of the current message is displayed in Rmail's mode line, followed by the total number of messages in the file. You can move to a message by specifying its message number with the `j` key (see Section 30.3 [Rmail Motion], page 633).

Following the usual conventions of Emacs, changes in an Rmail file become permanent only when you save the file. You can save it with `s` (`rmail-expunge-and-save`), which also expunges deleted messages from the file first (see Section 30.4 [Rmail Deletion], page 634). To save the file without expunging, use `C-x C-s`. Rmail also saves the Rmail file after merging new mail from an inbox file (see Section 30.5 [Rmail Inbox], page 635).

You can exit Rmail with `q` (`rmail-quit`); this expunges and saves the Rmail file, then buries the Rmail buffer as well as its summary buffer, if present (see Section 30.11 [Rmail Summary], page 642). But there is no need to "exit" formally. If you switch from Rmail to editing in other buffers, and never switch back, you have exited. Just make sure to save the Rmail file eventually (like any other file you have changed). `C-x s` is a suitable way to do this (see ⟨undefined⟩ [Save Commands], page ⟨undefined⟩). The Rmail command `b`, `rmail-bury`, buries the Rmail buffer and its summary without expunging and saving the Rmail file.

## 30.2 Scrolling Within a Message

When Rmail displays a message that does not fit on the screen, you must scroll through it to read the rest. You could do this with `C-v`, `M-v` and `M-<`, but in Rmail scrolling is so frequent that it deserves to be easier.

SPC Scroll forward (`scroll-up-command`).

DEL Scroll backward (`scroll-down-command`).

. Scroll to start of message (`rmail-beginning-of-message`).

`/` Scroll to end of message (`rmail-end-of-message`).

Since the most common thing to do while reading a message is to scroll through it by screenfuls, Rmail makes SPC and DEL do the same as `C-v` (`scroll-up-command`) and `M-v` (`scroll-down-command`) respectively.

The command `.` (`rmail-beginning-of-message`) scrolls back to the beginning of the selected message. This is not quite the same as `M-<`: for one thing, it does not set the mark; for another, it resets the buffer boundaries of the current message if you have changed them. Similarly, the command `/` (`rmail-end-of-message`) scrolls forward to the end of the selected message.

## 30.3 Moving Among Messages

The most basic thing to do with a message is to read it. The way to do this in Rmail is to make the message current. The usual practice is to move sequentially through the file, since this is the order of receipt of messages. When you enter Rmail, you are positioned at the first message that you have not yet made current (that is, the first one that has the '`unseen`' attribute; see Section 30.9 [Rmail Attributes], page 639). Move forward to see the other new messages; move backward to re-examine old messages.

`n` Move to the next nondeleted message, skipping any intervening deleted messages (`rmail-next-undeleted-message`).

`p` Move to the previous nondeleted message (`rmail-previous-undeleted-message`).

`M-n` Move to the next message, including deleted messages (`rmail-next-message`).

`M-p` Move to the previous message, including deleted messages (`rmail-previous-message`).

`C-c C-n` Move to the next message with the same subject as the current one (`rmail-next-same-subject`).

`C-c C-p` Move to the previous message with the same subject as the current one (`rmail-previous-same-subject`).

`j` Move to the first message. With argument *n*, move to message number *n* (`rmail-show-message`).

`>` Move to the last message (`rmail-last-message`).

`<` Move to the first message (`rmail-first-message`).

`M-s` *regexp* `RET`
Move to the next message containing a match for *regexp* (`rmail-search`).

`- M-s` *regexp* `RET`
Move to the previous message containing a match for *regexp*.

`n` and `p` are the usual way of moving among messages in Rmail. They move through the messages sequentially, but skip over deleted messages, which is usually what you want to do. Their command definitions are named `rmail-next-undeleted-message` and `rmail-previous-undeleted-message`. If you do not want to skip deleted messages—for example,

if you want to move to a message to undelete it—use the variants `M-n` and `M-p` (`rmail-next-message` and `rmail-previous-message`). A numeric argument to any of these commands serves as a repeat count.

In Rmail, you can specify a numeric argument by typing just the digits. You don't need to type `C-u` first.

The `M-s` (`rmail-search`) command is Rmail's version of search. The usual incremental search command `C-s` works in Rmail, but it searches only within the current message. The purpose of `M-s` is to search for another message. It reads a regular expression (see Section 12.6 [Regexps], page 206) nonincrementally, then searches starting at the beginning of the following message for a match. It then selects that message. If *regexp* is empty, `M-s` reuses the regexp used the previous time.

To search backward in the file for another message, give `M-s` a negative argument. In Rmail you can do this with `- M-s`. This begins searching from the end of the previous message.

It is also possible to search for a message based on labels. See Section 30.8 [Rmail Labels], page 638.

The `C-c C-n` (`rmail-next-same-subject`) command moves to the next message with the same subject as the current one. A prefix argument serves as a repeat count. With a negative argument, this command moves backward, acting like `C-c C-p` (`rmail-previous-same-subject`). When comparing subjects, these commands ignore the prefixes typically added to the subjects of replies.

To move to a message specified by absolute message number, use `j` (`rmail-show-message`) with the message number as argument. With no argument, `j` selects the first message. `<` (`rmail-first-message`) also selects the first message. `>` (`rmail-last-message`) selects the last message.

## 30.4 Deleting Messages

When you no longer need to keep a message, you can *delete* it. This flags it as ignorable, and some Rmail commands pretend it is no longer present; but it still has its place in the Rmail file, and still has its message number.

*Expunging* the Rmail file actually removes the deleted messages. The remaining messages are renumbered consecutively.

`d` Delete the current message, and move to the next nondeleted message (`rmail-delete-forward`).

`C-d` Delete the current message, and move to the previous nondeleted message (`rmail-delete-backward`).

`u` Undelete the current message, or move back to the previous deleted message and undelete it (`rmail-undelete-previous-message`).

`x` Expunge the Rmail file (`rmail-expunge`).

There are two Rmail commands for deleting messages. Both delete the current message and select another. `d` (`rmail-delete-forward`) moves to the following message, skipping messages already deleted, while `C-d` (`rmail-delete-backward`) moves to the previous nondeleted message. If there is no nondeleted message to move to in the specified direction, the

message that was just deleted remains current. `d` with a prefix argument is equivalent to `C-d`. Note that the Rmail summary versions of these commands behave slightly differently (see Section 30.11.2 [Rmail Summary Edit], page 643).

Whenever Rmail deletes a message, it runs the hook `rmail-delete-message-hook`. When the hook functions are invoked, the message has been marked deleted, but it is still the current message in the Rmail buffer.

To make all the deleted messages finally vanish from the Rmail file, type `x` (`rmail-expunge`). Until you do this, you can still *undelete* the deleted messages. The undeletion command, `u` (`rmail-undelete-previous-message`), is designed to cancel the effect of a `d` command in most cases. It undeletes the current message if the current message is deleted. Otherwise it moves backward to previous messages until a deleted message is found, and undeletes that message.

You can usually undo a `d` with a `u` because the `u` moves back to and undeletes the message that the `d` deleted. But this does not work when the `d` skips a few already-deleted messages that follow the message being deleted; then the `u` command undeletes the last of the messages that were skipped. There is no clean way to avoid this problem. However, by repeating the `u` command, you can eventually get back to the message that you intend to undelete. You can also select a particular deleted message with the `M-p` command, then type `u` to undelete it.

A deleted message has the '`deleted`' attribute, and as a result '`deleted`' appears in the mode line when the current message is deleted. In fact, deleting or undeleting a message is nothing more than adding or removing this attribute. See Section 30.9 [Rmail Attributes], page 639.

## 30.5 Rmail Files and Inboxes

When you receive mail locally, the operating system places incoming mail for you in a file that we call your *inbox*. When you start up Rmail, it runs a C program called `movemail` to copy the new messages from your local inbox into your primary Rmail file, which also contains other messages saved from previous Rmail sessions. It is in this file that you actually read the mail with Rmail. This operation is called *getting new mail*. You can get new mail at any time in Rmail by typing `g`.

The variable `rmail-primary-inbox-list` contains a list of the files that are inboxes for your primary Rmail file. If you don't set this variable explicitly, Rmail uses the `MAIL` environment variable, or, as a last resort, a default inbox based on `rmail-spool-directory`. The default inbox file depends on your operating system; often it is '`/var/mail/`*`username`*', '`/var/spool/mail/`*`username`*', or '`/usr/spool/mail/`*`username`*'.

You can specify the inbox file(s) for any Rmail file for the current session with the command `set-rmail-inbox-list`; see Section 30.6 [Rmail Files], page 636.

There are two reasons for having separate Rmail files and inboxes.

1. The inbox file format varies between operating systems and according to the other mail software in use. Only one part of Rmail needs to know about the alternatives, and it need only understand how to convert all of them to Rmail's own format.
2. It is very cumbersome to access an inbox file without danger of losing mail, because it is necessary to interlock with mail delivery. Moreover, different operating systems use

different interlocking techniques. The strategy of moving mail out of the inbox once and for all into a separate Rmail file avoids the need for interlocking in all the rest of Rmail, since only Rmail operates on the Rmail file.

Rmail was originally written to use the Babyl format as its internal format. Since then, we have recognized that the usual inbox format ('`mbox`') on Unix and GNU systems is adequate for the job, and so since Emacs 23 Rmail uses that as its internal format. The Rmail file is still separate from the inbox file, even though their format is the same.

When getting new mail, Rmail first copies the new mail from the inbox file to the Rmail file; then it saves the Rmail file; then it clears out the inbox file. This way, a system crash may cause duplication of mail between the inbox and the Rmail file, but cannot lose mail. If `rmail-preserve-inbox` is non-`nil`, then Rmail does not clear out the inbox file when it gets new mail. You may wish to set this, for example, on a portable computer you use to check your mail via POP while traveling, so that your mail will remain on the server and you can save it later on your workstation.

In some cases, Rmail copies the new mail from the inbox file indirectly. First it runs the `movemail` program to move the mail from the inbox to an intermediate file called '`.newmail-`*`inboxname`*', in the same directory as the Rmail file. Then Rmail merges the new mail from that file, saves the Rmail file, and only then deletes the intermediate file. If there is a crash at the wrong time, this file continues to exist, and Rmail will use it again the next time it gets new mail from that inbox.

If Rmail is unable to convert the data in '`.newmail-`*`inboxname`*' into mbox format, it renames the file to '`RMAILOSE.`*`n`*' (*n* is an integer chosen to make the name unique) so that Rmail will not have trouble with the data again. You should look at the file, find whatever message confuses Rmail (probably one that includes the control-underscore character, octal code 037), and delete it. Then you can use `1 g` to get new mail from the corrected file.

## 30.6 Multiple Rmail Files

Rmail operates by default on your *primary Rmail file*, which is named '`~/RMAIL`' and receives your incoming mail from your system inbox file. But you can also have other Rmail files and edit them with Rmail. These files can receive mail through their own inboxes, or you can move messages into them with explicit Rmail commands (see Section 30.7 [Rmail Output], page 637).

`i` *`file`* `RET`
: Read *file* into Emacs and run Rmail on it (`rmail-input`).

`M-x set-rmail-inbox-list RET` *`files`* `RET`
: Specify inbox file names for current Rmail file to get mail from.

`g`
: Merge new mail from current Rmail file's inboxes (`rmail-get-new-mail`).

`C-u g` *`file`* `RET`
: Merge new mail from inbox file *file*.

To run Rmail on a file other than your primary Rmail file, you can use the `i` (`rmail-input`) command in Rmail. This visits the file in Rmail mode. You can use `M-x rmail-input` even when not in Rmail, but it is easier to type `C-u M-x rmail`, which does the same thing.

The file you read with `i` should normally be a valid mbox file. If it is not, Rmail tries to convert its text to mbox format, and visits the converted text in the buffer. If you save the buffer, that converts the file.

If you specify a file name that doesn't exist, `i` initializes a new buffer for creating a new Rmail file.

You can also select an Rmail file from a menu. In the Classify menu, choose the Input Rmail File item; then choose the Rmail file you want. The variables `rmail-secondary-file-directory` and `rmail-secondary-file-regexp` specify which files to offer in the menu: the first variable says which directory to find them in; the second says which files in that directory to offer (all those that match the regular expression). If no files match, you cannot select this menu item. These variables also apply to choosing a file for output (see Section 30.7 [Rmail Output], page 637).

The inbox files to use are specified by the variable `rmail-inbox-list`, which is buffer-local in Rmail mode. As a special exception, if you have specified no inbox files for your primary Rmail file, it uses the `MAIL` environment variable, or your standard system inbox.

The `g` command (`rmail-get-new-mail`) merges mail into the current Rmail file from its inboxes. If the Rmail file has no inboxes, `g` does nothing. The command `M-x rmail` also merges new mail into your primary Rmail file.

To merge mail from a file that is not the usual inbox, give the `g` key a numeric argument, as in `C-u g`. Then it reads a file name and merges mail from that file. The inbox file is not deleted or changed in any way when `g` with an argument is used. This is, therefore, a general way of merging one file of messages into another.

## 30.7 Copying Messages Out to Files

These commands copy messages from an Rmail file into another file.

`o` *file* `RET`
: Append a full copy of the current message to the file *file* (`rmail-output`).

`C-o` *file* `RET`
: Append a copy of the current message, as displayed, to the file *file* (`rmail-output-as-seen`).

`w` *file* `RET`
: Output just the message body to the file *file*, taking the default file name from the message '`Subject`' header.

The commands `o` and `C-o` copy the current message into a specified file, adding it at the end. The two commands differ mainly in how much to copy: `o` copies the full message headers, even if they are not all visible, while `C-o` copies exactly the headers currently displayed and no more. See Section 30.13 [Rmail Display], page 645. In addition, `o` converts the message to Babyl format (used by Rmail in Emacs version 22 and before) if the file is in Babyl format; `C-o` cannot output to Babyl files at all.

If the output file is currently visited in an Emacs buffer, the output commands append the message to that buffer. It is up to you to save the buffer eventually in its file.

Sometimes you may receive a message whose body holds the contents of a file. You can save the body to a file (excluding the message header) with the `w` command (`rmail-output-body-to-file`). Often these messages contain the intended file name in the '`Subject`' field,

so the `w` command uses the '`Subject`' field as the default for the output file name. However, the file name is read using the minibuffer, so you can specify a different name if you wish.

You can also output a message to an Rmail file chosen with a menu. In the Classify menu, choose the Output Rmail File menu item; then choose the Rmail file you want. This outputs the current message to that file, like the `o` command. The variables `rmail-secondary-file-directory` and `rmail-secondary-file-regexp` specify which files to offer in the menu: the first variable says which directory to find them in; the second says which files in that directory to offer (all those that match the regular expression). If no files match, you cannot select this menu item.

Copying a message with `o` or `C-o` gives the original copy of the message the '`filed`' attribute, so that '`filed`' appears in the mode line when such a message is current.

If you like to keep just a single copy of every mail message, set the variable `rmail-delete-after-output` to `t`; then the `o`, `C-o` and `w` commands delete the original message after copying it. (You can undelete it afterward if you wish.)

The variable `rmail-output-file-alist` lets you specify intelligent defaults for the output file, based on the contents of the current message. The value should be a list whose elements have this form:

```
(regexp . name-exp)
```

If there's a match for *regexp* in the current message, then the default file name for output is *name-exp*. If multiple elements match the message, the first matching element decides the default file name. The subexpression *name-exp* may be a string constant giving the file name to use, or more generally it may be any Lisp expression that returns a file name as a string. `rmail-output-file-alist` applies to both `o` and `C-o`.

Rmail can automatically save messages from your primary Rmail file (the one that `rmail-file-name` specifies) to other files, based on the value of the variable `rmail-automatic-folder-directives`. This variable is a list of elements ('`directives`') that say which messages to save where. Each directive is a list consisting of an output file, followed by one or more pairs of a header name and a regular expression. If a message has a header matching the specified regular expression, that message is saved to the given file. If the directive has more than one header entry, all must match. Rmail checks directives when it shows a message from the file `rmail-file-name`, and applies the first that matches (if any). If the output file is `nil`, the message is deleted, not saved. For example, you can use this feature to save messages from a particular address, or with a particular subject, to a dedicated file.

## 30.8 Labels

Each message can have various *labels* assigned to it as a means of classification. Each label has a name; different names are different labels. Any given label is either present or absent on a particular message. A few label names have standard meanings and are given to messages automatically by Rmail when appropriate; these special labels are called *attributes*. All other labels are assigned only by users.

`a` *label* `RET`
: Assign the label *label* to the current message (`rmail-add-label`).

`k` *label* `RET`
: Remove the label *label* from the current message (`rmail-kill-label`).

`C-M-n` *labels* `RET`
: Move to the next message that has one of the labels *labels* (`rmail-next-labeled-message`).

`C-M-p` *labels* `RET`
: Move to the previous message that has one of the labels *labels* (`rmail-previous-labeled-message`).

`l` *labels* `RET`
`C-M-l` *labels* `RET`
: Make a summary of all messages containing any of the labels *labels* (`rmail-summary-by-labels`).

The `a` (`rmail-add-label`) and `k` (`rmail-kill-label`) commands allow you to assign or remove any label on the current message. If the *label* argument is empty, it means to assign or remove the same label most recently assigned or removed.

Once you have given messages labels to classify them as you wish, there are three ways to use the labels: in moving, in summaries, and in sorting.

`C-M-n` *labels* `RET` (`rmail-next-labeled-message`) moves to the next message that has one of the labels *labels*. The argument *labels* specifies one or more label names, separated by commas. `C-M-p` (`rmail-previous-labeled-message`) is similar, but moves backwards to previous messages. A numeric argument to either command serves as a repeat count.

The command `C-M-l` *labels* `RET` (`rmail-summary-by-labels`) displays a summary containing only the messages that have at least one of a specified set of labels. The argument *labels* is one or more label names, separated by commas. See Section 30.11 [Rmail Summary], page 642, for information on summaries.

If the *labels* argument to `C-M-n`, `C-M-p` or `C-M-l` is empty, it means to use the last set of labels specified for any of these commands.

See Section 30.12 [Rmail Sorting], page 645, for information on sorting messages with labels.

## 30.9 Rmail Attributes

Some labels such as '`deleted`' and '`filed`' have built-in meanings, and Rmail assigns them to messages automatically at appropriate times; these labels are called *attributes*. Here is a list of Rmail attributes:

'`unseen`'
: Means the message has never been current. Assigned to messages when they come from an inbox file, and removed when a message is made current. When you start Rmail, it initially shows the first message that has this attribute.

'`deleted`'
: Means the message is deleted. Assigned by deletion commands and removed by undeletion commands (see Section 30.4 [Rmail Deletion], page 634).

'`filed`'
: Means the message has been copied to some other file. Assigned by the `o` and `C-o` file output commands (see Section 30.7 [Rmail Output], page 637).

‘answered’
: Means you have mailed an answer to the message. Assigned by the `r` command (`rmail-reply`). See Section 30.10 [Rmail Reply], page 640.

‘forwarded’
: Means you have forwarded the message. Assigned by the `f` command (`rmail-forward`). See Section 30.10 [Rmail Reply], page 640.

‘edited’
: Means you have edited the text of the message within Rmail. See Section 30.15 [Rmail Editing], page 647.

‘resent’
: Means you have resent the message. Assigned by the command `M-x rmail-resend`. See Section 30.10 [Rmail Reply], page 640.

‘retried’
: Means you have retried a failed outgoing message. Assigned by the command `M-x rmail-retry-failure`. See Section 30.10 [Rmail Reply], page 640.

All other labels are assigned or removed only by users, and have no standard meaning.

## 30.10 Sending Replies

Rmail has several commands to send outgoing mail. See Chapter 29 [Sending Mail], page 624, for information on using Message mode, including certain features meant to work with Rmail. What this section documents are the special commands of Rmail for entering the mail buffer. Note that the usual keys for sending mail—`C-x m`, `C-x 4 m`, and `C-x 5 m`—also work normally in Rmail mode.

`m`
: Send a message (`rmail-mail`).

`c`
: Continue editing the already started outgoing message (`rmail-continue`).

`r`
: Send a reply to the current Rmail message (`rmail-reply`).

`f`
: Forward the current message to other users (`rmail-forward`).

`C-u f`
: Resend the current message to other users (`rmail-resend`).

`M-m`
: Try sending a bounced message a second time (`rmail-retry-failure`).

The most common reason to send a message while in Rmail is to reply to the message you are reading. To do this, type `r` (`rmail-reply`). This displays a mail composition buffer in another window, much like `C-x 4 m`, but preinitializes the ‘`Subject`’, ‘`To`’, ‘`CC`’, ‘`In-reply-to`’ and ‘`References`’ header fields based on the message you are replying to. The ‘`To`’ field starts out as the address of the person who sent the message you received, and the ‘`CC`’ field starts out with all the other recipients of that message.

You can exclude certain recipients from being included automatically in replies, using the variable `mail-dont-reply-to-names`. Its value should be a regular expression; any recipients that match are excluded from the ‘`CC`’ field. They are also excluded from the ‘`To`’ field, unless this would leave the field empty. If this variable is nil, then the first time you compose a reply it is initialized to a default value that matches your own address.

To omit the ‘`CC`’ field completely for a particular reply, enter the reply command with a numeric argument: `C-u r` or `1 r`. This means to reply only to the sender of the original message.

Once the mail composition buffer has been initialized, editing and sending the mail goes as usual (see Chapter 29 [Sending Mail], page 624). You can edit the presupplied header fields if they are not what you want. You can also use commands such as `C-c C-y`, which yanks in the message that you are replying to (see Section 29.4 [Mail Commands], page 627). You can also switch to the Rmail buffer, select a different message there, switch back, and yank the new current message.

Sometimes a message does not reach its destination. Mailers usually send the failed message back to you, enclosed in a *failure message*. The Rmail command `M-m` (`rmail-retry-failure`) prepares to send the same message a second time: it sets up a mail composition buffer with the same text and header fields as before. If you type `C-c C-c` right away, you send the message again exactly the same as the first time. Alternatively, you can edit the text or headers and then send it. The variable `rmail-retry-ignored-headers`, in the same format as `rmail-ignored-headers` (see Section 30.13 [Rmail Display], page 645), controls which headers are stripped from the failed message when retrying it.

Another frequent reason to send mail in Rmail is to *forward* the current message to other users. `f` (`rmail-forward`) makes this easy by preinitializing the mail composition buffer with the current message as the text, and a subject of the form `[`*`from`*`: `*`subject`*`]`, where *from* and *subject* are the sender and subject of the original message. All you have to do is fill in the recipients and send. When you forward a message, recipients get a message which is "from" you, and which has the original message in its contents.

Rmail offers two formats for forwarded messages. The default is to use MIME (see Section 30.13 [Rmail Display], page 645) format. This includes the original message as a separate part. You can use a simpler format if you prefer, by setting the variable `rmail-enable-mime-composing` to `nil`. In this case, Rmail just includes the original message enclosed between two delimiter lines. It also modifies every line that starts with a dash, by inserting '`- `' at the start of the line. When you receive a forwarded message in this format, if it contains something besides ordinary text—for example, program source code—you might find it useful to undo that transformation. You can do this by selecting the forwarded message and typing `M-x unforward-rmail-message`. This command extracts the original forwarded message, deleting the inserted '`- `' strings, and inserts it into the Rmail file as a separate message immediately following the current one.

*Resending* is an alternative similar to forwarding; the difference is that resending sends a message that is "from" the original sender, just as it reached you—with a few added header fields ('`Resent-From`' and '`Resent-To`') to indicate that it came via you. To resend a message in Rmail, use `C-u f`. (`f` runs `rmail-forward`, which invokes `rmail-resend` if you provide a numeric argument.)

Use the `m` (`rmail-mail`) command to start editing an outgoing message that is not a reply. It leaves the header fields empty. Its only difference from `C-x 4 m` is that it makes the Rmail buffer accessible for `C-c C-y`, just as `r` does.

The `c` (`rmail-continue`) command resumes editing the mail composition buffer, to finish editing an outgoing message you were already composing, or to alter a message you have sent.

If you set the variable `rmail-mail-new-frame` to a non-`nil` value, then all the Rmail commands to start sending a message create a new frame to edit it in. This frame is deleted when you send the message.

All the Rmail commands to send a message use the mail-composition method that you have chosen (see Section 29.7 [Mail Methods], page 631).

## 30.11 Summaries

A *summary* is a buffer containing one line per message to give you an overview of the mail in an Rmail file. Each line shows the message number and date, the sender, the line count, the labels, and the subject. Moving point in the summary buffer selects messages as you move to their summary lines. Almost all Rmail commands are valid in the summary buffer also; when used there, they apply to the message described by the current line of the summary.

A summary buffer applies to a single Rmail file only; if you are editing multiple Rmail files, each one can have its own summary buffer. The summary buffer name is made by appending '`-summary`' to the Rmail buffer's name. Normally only one summary buffer is displayed at a time.

### 30.11.1 Making Summaries

Here are the commands to create a summary for the current Rmail buffer. Once the Rmail buffer has a summary, changes in the Rmail buffer (such as deleting or expunging messages, and getting new mail) automatically update the summary.

`h`
`C-M-h` Summarize all messages (`rmail-summary`).

`l` *labels* `RET`
`C-M-l` *labels* `RET`
Summarize messages that have one or more of the specified labels (`rmail-summary-by-labels`).

`C-M-r` *rcpts* `RET`
Summarize messages that match the specified recipients (`rmail-summary-by-recipients`).

`C-M-t` *topic* `RET`
Summarize messages that have a match for the specified regexp *topic* in their subjects (`rmail-summary-by-topic`).

`C-M-s` *regexp* `RET`
Summarize messages whose headers match the specified regular expression *regexp* (`rmail-summary-by-regexp`).

`C-M-f` *senders* `RET`
Summarize messages that match the specified senders. (`rmail-summary-by-senders`).

The `h` or `C-M-h` (`rmail-summary`) command fills the summary buffer for the current Rmail buffer with a summary of all the messages in the buffer. It then displays and selects the summary buffer in another window.

`C-M-l` *labels* `RET` (`rmail-summary-by-labels`) makes a partial summary mentioning only the messages that have one or more of the labels *labels*. *labels* should contain label names separated by commas.

`C-M-r` *rcpts* `RET` (`rmail-summary-by-recipients`) makes a partial summary mentioning only the messages that have one or more recipients matching the regular expression *rcpts*. You can use commas to separate multiple regular expressions. These are matched against the '`To`', '`From`', and '`CC`' headers (supply a prefix argument to exclude this header).

`C-M-t` *topic* `RET` (`rmail-summary-by-topic`) makes a partial summary mentioning only the messages whose subjects have a match for the regular expression *topic*. You can use commas to separate multiple regular expressions. With a prefix argument, the match is against the whole message, not just the subject.

`C-M-s` *regexp* `RET` (`rmail-summary-by-regexp`) makes a partial summary that mentions only the messages whose headers (including the date and the subject lines) match the regular expression *regexp*.

`C-M-f` *senders* `RET` (`rmail-summary-by-senders`) makes a partial summary that mentions only the messages whose '`From`' fields match the regular expression *senders*. You can use commas to separate multiple regular expressions.

Note that there is only one summary buffer for any Rmail buffer; making any kind of summary discards any previous summary.

The variable `rmail-summary-window-size` says how many lines to use for the summary window. The variable `rmail-summary-line-count-flag` controls whether the summary line for a message should include the line count of the message. Setting this option to nil might speed up the generation of summaries.

### 30.11.2 Editing in Summaries

You can use the Rmail summary buffer to do almost anything you can do in the Rmail buffer itself. In fact, once you have a summary buffer, there's no need to switch back to the Rmail buffer.

You can select and display various messages in the Rmail buffer, from the summary buffer, just by moving point in the summary buffer to different lines. It doesn't matter what Emacs command you use to move point; whichever line point is on at the end of the command, that message is selected in the Rmail buffer.

Almost all Rmail commands work in the summary buffer as well as in the Rmail buffer. Thus, `d` in the summary buffer deletes the current message, `u` undeletes, and `x` expunges. (However, in the summary buffer, a numeric argument to `d`, `C-d` and `u` serves as a repeat count. A negative argument reverses the meaning of `d` and `C-d`. Also, if there are no more undeleted messages in the relevant direction, the delete commands go to the first or last message, rather than staying on the current message.) `o` and `C-o` output the current message to a FILE; `r` starts a reply to it; etc. You can scroll the current message while remaining in the summary buffer using SPC and DEL.

`M-u` (`rmail-summary-undelete-many`) undeletes all deleted messages in the summary. A prefix argument means to undelete that many of the previous deleted messages.

The Rmail commands to move between messages also work in the summary buffer, but with a twist: they move through the set of messages included in the summary. They also ensure the Rmail buffer appears on the screen (unlike cursor motion commands, which update the contents of the Rmail buffer but don't display it in a window unless it already appears). Here is a list of these commands:

`n` Move to next line, skipping lines saying 'deleted', and select its message (`rmail-summary-next-msg`).

`p` Move to previous line, skipping lines saying 'deleted', and select its message (`rmail-summary-previous-msg`).

`M-n` Move to next line and select its message (`rmail-summary-next-all`).

`M-p` Move to previous line and select its message (`rmail-summary-previous-all`).

`>` Move to the last line, and select its message (`rmail-summary-last-message`).

`<` Move to the first line, and select its message (`rmail-summary-first-message`).

`j`
RET Select the message on the current line (ensuring that the Rmail buffer appears on the screen; `rmail-summary-goto-msg`). With argument *n*, select message number *n* and move to its line in the summary buffer; this signals an error if the message is not listed in the summary buffer.

`M-s` *pattern* RET
Search through messages for *pattern* starting with the current message; select the message found, and move point in the summary buffer to that message's line (`rmail-summary-search`). A prefix argument acts as a repeat count; a negative argument means search backward (equivalent to `rmail-summary-search-backward`.)

`C-M-n` *labels* RET
Move to the next message with at least one of the specified labels (`rmail-summary-next-labeled-message`). *labels* is a comma-separated list of labels. A prefix argument acts as a repeat count.

`C-M-p` *labels* RET
Move to the previous message with at least one of the specified labels (`rmail-summary-previous-labeled-message`).

`C-c C-n` RET
Move to the next message with the same subject as the current message (`rmail-summary-next-same-subject`). A prefix argument acts as a repeat count.

`C-c C-p` RET
Move to the previous message with the same subject as the current message (`rmail-summary-previous-same-subject`).

Deletion, undeletion, and getting new mail, and even selection of a different message all update the summary buffer when you do them in the Rmail buffer. If the variable `rmail-redisplay-summary` is non-`nil`, these actions also bring the summary buffer back onto the screen.

When you are finished using the summary, type `Q` (`rmail-summary-wipe`) to delete the summary buffer's window. You can also exit Rmail while in the summary: `q` (`rmail-summary-quit`) deletes the summary window, then exits from Rmail by saving the Rmail file and switching to another buffer. Alternatively, `b` (`rmail-summary-bury`) simply buries the Rmail summary and buffer.

## 30.12 Sorting the Rmail File

`C-c C-s C-d`
`M-x rmail-sort-by-date`
: Sort messages of current Rmail buffer by date.

`C-c C-s C-s`
`M-x rmail-sort-by-subject`
: Sort messages of current Rmail buffer by subject.

`C-c C-s C-a`
`M-x rmail-sort-by-author`
: Sort messages of current Rmail buffer by author's name.

`C-c C-s C-r`
`M-x rmail-sort-by-recipient`
: Sort messages of current Rmail buffer by recipient's names.

`C-c C-s C-c`
`M-x rmail-sort-by-correspondent`
: Sort messages of current Rmail buffer by the name of the other correspondent.

`C-c C-s C-l`
`M-x rmail-sort-by-lines`
: Sort messages of current Rmail buffer by number of lines.

`C-c C-s C-k RET` *`labels`* `RET`
`M-x rmail-sort-by-labels RET` *`labels`* `RET`
: Sort messages of current Rmail buffer by labels. The argument *labels* should be a comma-separated list of labels. The order of these labels specifies the order of messages; messages with the first label come first, messages with the second label come second, and so on. Messages that have none of these labels come last.

The Rmail sort commands perform a *stable sort*: if there is no reason to prefer either one of two messages, their order remains unchanged. You can use this to sort by more than one criterion. For example, if you use `rmail-sort-by-date` and then `rmail-sort-by-author`, messages from the same author appear in order by date.

With a prefix argument, all these commands reverse the order of comparison. This means they sort messages from newest to oldest, from biggest to smallest, or in reverse alphabetical order.

The same keys in the summary buffer run similar functions; for example, `C-c C-s C-l` runs `rmail-summary-sort-by-lines`. Note that these commands always sort the whole Rmail buffer, even if the summary is only showing a subset of messages.

Note that you cannot undo a sort, so you may wish to save the Rmail buffer before sorting it.

## 30.13 Display of Messages

This section describes how Rmail displays mail headers, MIME sections and attachments, URLs, and encrypted messages.

`t` Toggle display of complete header (`rmail-toggle-header`).

Before displaying each message for the first time, Rmail reformats its header, hiding uninteresting header fields to reduce clutter. The `t` (`rmail-toggle-header`) command toggles this, switching between showing the reformatted header fields and showing the complete, original header. With a positive prefix argument, the command shows the reformatted header; with a zero or negative prefix argument, it shows the full header. Selecting the message again also reformats it if necessary.

The variable `rmail-ignored-headers` holds a regular expression specifying the header fields to hide; any matching header line will be hidden. The variable `rmail-nonignored-headers` overrides this: any header field matching that regular expression is shown even if it matches `rmail-ignored-headers` too. The variable `rmail-displayed-headers` is an alternative to these two variables; if non-`nil`, this should be a regular expression specifying which headers to display (the default is `nil`).

Rmail highlights certain header fields that are especially interesting—by default, the '`From`' and '`Subject`' fields. This highlighting uses the `rmail-highlight` face. The variable `rmail-highlighted-headers` holds a regular expression specifying the header fields to highlight; if it matches the beginning of a header field, that whole field is highlighted. To disable this feature, set `rmail-highlighted-headers` to `nil`.

If a message is in MIME (Multipurpose Internet Mail Extensions) format and contains multiple parts (MIME entities), Rmail displays each part with a *tagline*. The tagline summarizes the part's index, size, and content type. Depending on the content type, it may also contain one or more buttons; these perform actions such as saving the part into a file.

RET Hide or show the MIME part at point (`rmail-mime-toggle-hidden`).

TAB Move point to the next MIME tagline button. (`rmail-mime-next-item`).

`S-TAB` Move point to the previous MIME part (`rmail-mime-previous-item`).

`v` Toggle between MIME display and raw message (`rmail-mime`).

Each plain-text MIME part is initially displayed immediately after its tagline, as part of the Rmail buffer, while MIME parts of other types are represented only by their taglines, with their actual contents hidden. In either case, you can toggle a MIME part between its "displayed" and "hidden" states by typing RET anywhere in the part—or anywhere in its tagline (except for buttons for other actions, if there are any). Type RET (or click with the mouse) to activate a tagline button, and TAB to cycle point between tagline buttons.

The `v` (`rmail-mime`) command toggles between the default MIME display described above, and a "raw" display showing the undecoded MIME data. With a prefix argument, this command toggles the display of only an entity at point.

To prevent Rmail from handling MIME decoded messages, change the variable `rmail-enable-mime` to `nil`. When this is the case, the `v` (`rmail-mime`) command instead creates a temporary buffer to display the current MIME message.

If the current message is an encrypted one, use the command `M-x rmail-epa-decrypt` to decrypt it, using the EasyPG library (see Section "EasyPG" in *EasyPG Assistant User's Manual*).

You can highlight and activate URLs in the Rmail buffer using Goto Address mode:

```
(add-hook 'rmail-show-message-hook 'goto-address-mode)
```

Then you can browse these URLs by clicking on them with `Mouse-2` (or `Mouse-1` quickly) or by moving to one and typing `C-c RET`. See Section 31.11.2 [Activating URLs], page 678.

## 30.14 Rmail and Coding Systems

Rmail automatically decodes messages which contain non-ASCII characters, just as Emacs does with files you visit and with subprocess output. Rmail uses the standard '`charset=`*charset*' header in the message, if any, to determine how the message was encoded by the sender. It maps *charset* into the corresponding Emacs coding system (see Section 19.6 [Coding Systems], page 381), and uses that coding system to decode message text. If the message header doesn't have the '`charset`' specification, or if *charset* is not recognized, Rmail chooses the coding system with the usual Emacs heuristics and defaults (see Section 19.7 [Recognize Coding], page 383).

Occasionally, a message is decoded incorrectly, either because Emacs guessed the wrong coding system in the absence of the '`charset`' specification, or because the specification was inaccurate. For example, a misconfigured mailer could send a message with a '`charset=iso-8859-1`' header when the message is actually encoded in `koi8-r`. When you see the message text garbled, or some of its characters displayed as hex codes or empty boxes, this may have happened.

You can correct the problem by decoding the message again using the right coding system, if you can figure out or guess which one is right. To do this, invoke the `M-x rmail-redecode-body` command. It reads the name of a coding system, and then redecodes the message using the coding system you specified. If you specified the right coding system, the result should be readable.

## 30.15 Editing Within a Message

Most of the usual Emacs key bindings are available in Rmail mode, though a few, such as `C-M-n` and `C-M-h`, are redefined by Rmail for other purposes. However, the Rmail buffer is normally read only, and most of the letters are redefined as Rmail commands. If you want to edit the text of a message, you must use the Rmail command `e`.

`e` Edit the current message as ordinary text.

The `e` command (`rmail-edit-current-message`) switches from Rmail mode into Rmail Edit mode, another major mode which is nearly the same as Text mode. The mode line indicates this change.

In Rmail Edit mode, letters insert themselves as usual and the Rmail commands are not available. You can edit the message body and header fields. When you are finished editing the message, type `C-c C-c` to switch back to Rmail mode. Alternatively, you can return to Rmail mode but cancel any editing that you have done, by typing `C-c C-]`.

Entering Rmail Edit mode runs the hook `text-mode-hook`; then it runs the hook `rmail-edit-mode-hook` (see Section 33.2.2 [Hooks], page 696). Returning to ordinary Rmail mode adds the attribute '`edited`' to the message, if you have made any changes in it.

## 30.16 Digest Messages

A *digest message* is a message which exists to contain and carry several other messages. Digests are used on some mailing lists; all the messages that arrive for the list during a period of time such as one day are put inside a single digest which is then sent to the subscribers. Transmitting the single digest uses less computer time than transmitting the individual messages even though the total size is the same, because of the per-message overhead in network mail transmission.

When you receive a digest message, the most convenient way to read it is to *undigestify* it: to turn it back into many individual messages. Then you can read and delete the individual messages as it suits you. To do this, select the digest message and type the command `M-x undigestify-rmail-message`. This extracts the submessages as separate Rmail messages, and inserts them following the digest. The digest message itself is flagged as deleted.

## 30.17 Reading Rot13 Messages

Mailing list messages that might offend or annoy some readers are sometimes encoded in a simple code called *rot13*—so named because it rotates the alphabet by 13 letters. This code is not for secrecy, as it provides none; rather, it enables those who wish to to avoid seeing the real text of the message. For example, a review of a film might use rot13 to hide important plot points.

To view a buffer that uses the rot13 code, use the command `M-x rot13-other-window`. This displays the current buffer in another window which applies the code when displaying the text.

## 30.18 `movemail` program

Rmail uses the `movemail` program to move mail from your inbox to your Rmail file (see Section 30.5 [Rmail Inbox], page 635). When loaded for the first time, Rmail attempts to locate the `movemail` program and determine its version. There are two versions of the `movemail` program: the native one, shipped with GNU Emacs (the "emacs version") and the one included in GNU mailutils (the "mailutils version", see Section "movemail" in *GNU mailutils*). They support the same command line syntax and the same basic subset of options. However, the Mailutils version offers additional features.

The Emacs version of `movemail` is able to retrieve mail from the usual Unix mailbox formats and from remote mailboxes using the POP3 protocol.

The Mailutils version is able to handle a wide set of mailbox formats, such as plain Unix mailboxes, `maildir` and `MH` mailboxes, etc. It is able to access remote mailboxes using the POP3 or IMAP4 protocol, and can retrieve mail from them using a TLS encrypted channel. It also accepts mailbox arguments in URL form. The detailed description of mailbox URLs can be found in Section "URL" in *Mailbox URL Formats*. In short, a URL is:

```
proto://[user[:password]@]host-or-file-name
```

where square brackets denote optional elements.

*proto* Specifies the *mailbox protocol*, or *format* to use. The exact semantics of the rest of URL elements depends on the actual value of *proto* (see below).

*user* User name to access the remote mailbox.

*password* User password to access the remote mailbox.

*host-or-file-name*
Hostname of the remote server for remote mailboxes or file name of a local mailbox.

*Proto* can be one of:

`mbox` Usual Unix mailbox format. In this case, neither *user* nor *pass* are used, and *host-or-file-name* denotes the file name of the mailbox file, e.g., `mbox://var/spool/mail/smith`.

`mh` A local mailbox in the MH format. *User* and *pass* are not used. *Host-or-file-name* denotes the name of MH folder, e.g., `mh://Mail/inbox`.

`maildir` A local mailbox in the maildir format. *User* and *pass* are not used, and *host-or-file-name* denotes the name of `maildir` mailbox, e.g., `maildir://mail/inbox`.

`file` Any local mailbox format. Its actual format is detected automatically by `movemail`.

`pop` A remote mailbox to be accessed via POP3 protocol. *User* specifies the remote user name to use, *pass* may be used to specify the user password, *host-or-file-name* is the name or IP address of the remote mail server to connect to; e.g., `pop://smith:guessme@remote.server.net`.

`imap` A remote mailbox to be accessed via IMAP4 protocol. *User* specifies the remote user name to use, *pass* may be used to specify the user password, *host-or-file-name* is the name or IP address of the remote mail server to connect to; e.g., `imap://smith:guessme@remote.server.net`.

Alternatively, you can specify the file name of the mailbox to use. This is equivalent to specifying the '`file`' protocol:

`/var/spool/mail/`*user* ≡ `file://var/spool/mail/`*user*

The variable `rmail-movemail-program` controls which version of `movemail` to use. If that is a string, it specifies the absolute file name of the `movemail` executable. If it is `nil`, Rmail searches for `movemail` in the directories listed in `rmail-movemail-search-path`, then in `exec-path` (see Section 31.3 [Shell], page 655), then in `exec-directory`.

## 30.19 Retrieving Mail from Remote Mailboxes

Some sites use a method called POP for accessing users' inbox data instead of storing the data in inbox files. By default, the `Emacs movemail` can work with POP (unless the Emacs `configure` script was run with the option '`--without-pop`').

Similarly, the Mailutils `movemail` by default supports POP, unless it was configured with the '`--disable-pop`' option.

Both versions of `movemail` only work with POP3, not with older versions of POP.

No matter which flavor of `movemail` you use, you can specify a POP inbox by using a POP *URL* (see Section 30.18 [Movemail], page 648). A POP URL is a "file name" of the form '`pop://`*username*`@`*hostname*', where *hostname* is the host name or IP address of the

remote mail server and *username* is the user name on that server. Additionally, you may specify the password in the mailbox URL: '`pop://username:password@hostname`'. In this case, *password* takes preference over the one set by `rmail-remote-password` (see below). This is especially useful if you have several remote mailboxes with different passwords.

For backward compatibility, Rmail also supports an alternative way of specifying remote POP mailboxes. Specifying an inbox name in the form '`po:username:hostname`' is equivalent to '`pop://username@hostname`'. If you omit the *:hostname* part, the `MAILHOST` environment variable specifies the machine on which to look for the POP server.

Another method for accessing remote mailboxes is IMAP. This method is supported only by the Mailutils `movemail`. To specify an IMAP mailbox in the inbox list, use the following mailbox URL: '`imap://username[:password]@hostname`'. The *password* part is optional, as described above.

Accessing a remote mailbox may require a password. Rmail uses the following algorithm to retrieve it:

1. If a *password* is present in the mailbox URL (see above), it is used.
2. If the variable `rmail-remote-password-required` is `nil`, Rmail assumes no password is required.
3. If the variable `rmail-remote-password` is non-`nil`, its value is used.
4. Otherwise, Rmail will ask you for the password to use.

If you need to pass additional command-line flags to `movemail`, set the variable `rmail-movemail-flags` a list of the flags you wish to use. Do not use this variable to pass the '`-p`' flag to preserve your inbox contents; use `rmail-preserve-inbox` instead.

The `movemail` program installed at your site may support Kerberos authentication (the Emacs `movemail` does so if Emacs was configured with the option `--with-kerberos` or `--with-kerberos5`). If it is supported, it is used by default whenever you attempt to retrieve POP mail when `rmail-remote-password` and `rmail-remote-password-required` are unset.

Some POP servers store messages in reverse order. If your server does this, and you would rather read your mail in the order in which it was received, you can tell `movemail` to reverse the order of downloaded messages by adding the '`-r`' flag to `rmail-movemail-flags`.

Mailutils `movemail` supports TLS encryption. If you wish to use it, add the '`--tls`' flag to `rmail-movemail-flags`.

## 30.20 Retrieving Mail from Local Mailboxes in Various Formats

If your incoming mail is stored on a local machine in a format other than Unix mailbox, you will need the Mailutils `movemail` to retrieve it. See Section 30.18 [Movemail], page 648, for the detailed description of `movemail` versions. For example, to access mail from a inbox in `maildir` format located in '`/var/spool/mail/in`', you would include the following in the Rmail inbox list:

```
maildir://var/spool/mail/in
```

# 31 Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else: viewing "document files", reading Usenet news, running shell commands and shell subprocesses, using a single shared Emacs for utilities that expect to run an editor as a subprocess, printing hardcopy, sorting text, narrowing display to part of the buffer, editing binary files, saving an Emacs session for later resumption, following hyperlinks, browsing images, emulating other editors, and various diversions and amusements.

## 31.1 Gnus

Gnus is an Emacs package primarily designed for reading and posting Usenet news. It can also be used to read and respond to messages from a number of other sources—email, remote directories, digests, and so on. Here we introduce Gnus and describe several basic features. For full details on Gnus, type `C-h i` and then select the Gnus manual.

### 31.1.1 Gnus Buffers

Gnus uses several buffers to display information and to receive commands. The three most commonly-used Gnus buffers are the *group buffer*, the *summary buffer* and the *article buffer*.

The *group buffer* contains a list of article sources (e.g., newsgroups and email inboxes), which are collectively referred to as *groups*. This is the first buffer Gnus displays when it starts up. It normally displays only the groups to which you subscribe and that contain unread articles. From this buffer, you can select a group to read.

The *summary buffer* lists the articles in a single group, showing one article per line. By default, it displays each article's author, subject, and line number. The summary buffer is created when you select a group in the group buffer, and is killed when you exit the group.

From the summary buffer, you can choose an article to view. The article is displayed in the *article buffer*. In normal Gnus usage, you view this buffer but do not select it—all useful Gnus commands can be invoked from the summary buffer. But you can select the article buffer, and execute Gnus commands from it, if you wish.

### 31.1.2 When Gnus Starts Up

If your system has been set up for reading Usenet news, getting started with Gnus is easy—just type `M-x gnus`.

On starting up, Gnus reads your *news initialization file*: a file named '`.newsrc`' in your home directory which lists your Usenet newsgroups and subscriptions (this file is not unique to Gnus; it is used by many other newsreader programs). It then tries to contact the system's default news server, which is typically specified by the `NNTPSERVER` environment variable.

If your system does not have a default news server, or if you wish to use Gnus for reading email, then before invoking `M-x gnus` you need to tell Gnus where to get news and/or mail. To do this, customize the variables `gnus-select-method` and/or `gnus-secondary-select-methods`. See the Gnus manual for details.

Once Gnus has started up, it displays the group buffer. By default, the group buffer shows only a small number of *subscribed groups*. Groups with other statuses—*unsubscribed*, *killed*, or *zombie*—are hidden. The first time you start Gnus, any group to which you are

not subscribed is made into a killed group; any group that subsequently appears on the news server becomes a zombie group.

To proceed, you must select a group in the group buffer to open the summary buffer for that group; then, select an article in the summary buffer to view its article buffer in a separate window. The following sections explain how to use the group and summary buffers to do this.

To quit Gnus, type `q` in the group buffer. This automatically records your group statuses in the files '`.newsrc`' and '`.newsrc.eld`', so that they take effect in subsequent Gnus sessions.

### 31.1.3 Using the Gnus Group Buffer

The following commands are available in the Gnus group buffer:

SPC — Switch to the summary buffer for the group on the current line.

`l`
`A s` — In the group buffer, list only the groups to which you subscribe and which contain unread articles (this is the default listing).

`L`
`A u` — List all subscribed and unsubscribed groups, but not killed or zombie groups.

`A k` — List killed groups.

`A z` — List zombie groups.

`u` — Toggle the subscription status of the group on the current line (i.e., turn a subscribed group into an unsubscribed group, or vice versa). Invoking this on a killed or zombie group turns it into an unsubscribed group.

`C-k` — Kill the group on the current line. Killed groups are not recorded in the '`.newsrc`' file, and they are not shown in the `l` or `L` listings.

DEL — Move point to the previous group containing unread articles.

`n` — Move point to the next unread group.

`p` — Move point to the previous unread group.

`q` — Update your Gnus settings, and quit Gnus.

### 31.1.4 Using the Gnus Summary Buffer

The following commands are available in the Gnus summary buffer:

SPC — If there is no article selected, select the article on the current line and display its article buffer. Otherwise, try scrolling the selected article buffer in its window; on reaching the end of the buffer, select the next unread article.

Thus, you can read through all articles by repeatedly typing SPC.

DEL — Scroll the text of the article backwards.

`n` — Select the next unread article.

`p` — Select the previous unread article.

`s` Do an incremental search on the selected article buffer, as if you switched to the buffer and typed `C-s` (see Section 12.1 [Incremental Search], page 199).

`M-s` *regexp* `RET`
Search forward for articles containing a match for *regexp*.

`q` Exit the summary buffer and return to the group buffer.

## 31.2 Document Viewing

DocView mode is a major mode for viewing DVI, PostScript (PS), PDF, OpenDocument, and Microsoft Office documents. It provides features such as slicing, zooming, and searching inside documents. It works by converting the document to a set of images using the `gs` (GhostScript) command and other external tools[1], and displaying those images.

When you visit a document file that can be displayed with DocView mode, Emacs automatically uses DocView mode[2]. As an exception, when you visit a PostScript file, Emacs switches to PS mode, a major mode for editing PostScript files as text; however, it also enables DocView minor mode, so you can type `C-c C-c` to view the document with DocView. In either DocView mode or DocView minor mode, repeating `C-c C-c` (`doc-view-toggle-display`) toggles between DocView and the underlying file contents.

You can explicitly enable DocView mode with the command `M-x doc-view-mode`. You can toggle DocView minor mode with `M-x doc-view-minor-mode`.

When DocView mode starts, it displays a welcome screen and begins formatting the file, page by page. It displays the first page once that has been formatted.

To kill the DocView buffer, type `k` (`doc-view-kill-proc-and-buffer`). To bury it, type `q` (`quit-window`).

### 31.2.1 DocView Navigation

In DocView mode, you can scroll the current page using the usual Emacs movement keys: `C-p`, `C-n`, `C-b`, `C-f`, and the arrow keys.

By default, the line-motion keys `C-p` and `C-n` stop scrolling at the beginning and end of the current page, respectively. However, if you change the variable `doc-view-continuous` to a non-`nil` value, then `C-p` displays the previous page if you are already at the beginning of the current page, and `C-n` displays the next page if you are at the end of the current page.

You can also display the next page by typing `n`, NEXT or `C-x ]` (`doc-view-next-page`). To display the previous page, type `p`, PRIOR or `C-x [` (`doc-view-previous-page`).

SPC (`doc-view-scroll-up-or-next-page`) is a convenient way to advance through the document. It scrolls within the current page or advances to the next. DEL moves backwards in a similar way (`doc-view-scroll-down-or-previous-page`).

---

[1] `gs` is a hard requirement. For DVI files, `dvipdf` or `dvipdfm` is needed. For OpenDocument and Microsoft Office documents, the `unoconv` tool is needed.

[2] The needed external tools for the document type must be available, and Emacs must be running in a graphical frame and have PNG image support. If any of these requirements is not fulfilled, Emacs falls back to another major mode.

To go to the first page, type `M-<` (`doc-view-first-page`); to go to the last one, type `M->` (`doc-view-last-page`). To jump to a page by its number, type `M-g M-g` or `M-g g` (`doc-view-goto-page`).

You can enlarge or shrink the document with `+` (`doc-view-enlarge`) and `-` (`doc-view-shrink`). These commands work by reconverting the document at the new size. To specify the default size for DocView, customize the variable `doc-view-resolution`.

### 31.2.2 DocView Searching

In DocView mode, you can search the file's text for a regular expression (see Section 12.6 [Regexps], page 206). The interface for searching is inspired by `isearch` (see Section 12.1 [Incremental Search], page 199).

To begin a search, type `C-s` (`doc-view-search`) or `C-r` (`doc-view-search-backward`). This reads a regular expression using a minibuffer, then echoes the number of matches found within the document. You can move forward and back among the matches by typing `C-s` and `C-r`. DocView mode has no way to show the match inside the page image; instead, it displays a tooltip (at the mouse position) listing all matching lines in the current page. To force display of this tooltip, type `C-t` (`doc-view-show-tooltip`).

To start a new search, use the search command with a prefix argument; i.e., `C-u C-s` for a forward search or `C-u C-r` for a backward search.

### 31.2.3 DocView Slicing

Documents often have wide margins for printing. They are annoying when reading the document on the screen, because they use up screen space and can cause inconvenient scrolling.

With DocView you can hide these margins by selecting a *slice* of pages to display. A slice is a rectangle within the page area; once you specify a slice in DocView, it applies to whichever page you look at.

To specify the slice numerically, type `s s` (`doc-view-set-slice`); then enter the top left pixel position and the slice's width and height.

A more convenient graphical way to specify the slice is with `s m` (`doc-view-set-slice-using-mouse`), where you use the mouse to select the slice.

The most convenient way is to set the optimal slice by using BoundingBox information automatically determined from the document by typing `s b` (`doc-view-set-slice-using-mouse`).

To cancel the selected slice, type `s r` (`doc-view-reset-slice`). Then DocView shows the entire page including its entire margins.

### 31.2.4 DocView Conversion

For efficiency, DocView caches the images produced by `gs`. The name of this directory is given by the variable `doc-view-cache-directory`. You can clear the cache directory by typing `M-x doc-view-clear-cache`.

To force reconversion of the currently viewed document, type `r` or `g` (`revert-buffer`). To kill the converter process associated with the current buffer, type `K` (`doc-view-kill-proc`). The command `k` (`doc-view-kill-proc-and-buffer`) kills the converter process and the DocView buffer.

## 31.3 Running Shell Commands from Emacs

Emacs has commands for passing single command lines to shell subprocesses, and for running a shell interactively with input and output to an Emacs buffer, and for running a shell in a terminal emulator window.

`M-!` *cmd* `RET`
: Run the shell command *cmd* and display the output (`shell-command`).

`M-|` *cmd* `RET`
: Run the shell command *cmd* with region contents as input; optionally replace the region with the output (`shell-command-on-region`).

`M-&` *cmd* `RET`
: Run the shell command *cmd* asynchronously, and display the output (`async-shell-command`).

`M-x shell`
: Run a subshell with input and output through an Emacs buffer. You can then give commands interactively.

`M-x term`
: Run a subshell with input and output through an Emacs buffer. You can then give commands interactively. Full terminal emulation is available.

Whenever you specify a relative file name for an executable program (either in the *cmd* argument to one of the above commands, or in other contexts), Emacs searches for the program in the directories specified by the variable `exec-path`. The value of this variable must be a list of directory names; the default value is initialized from the environment variable `PATH` when Emacs is started (see Section E.4.1 [General Variables], page 758).

`M-x eshell` invokes a shell implemented entirely in Emacs. It is documented in its own manual. See the Eshell Info manual, which is distributed with Emacs.

### 31.3.1 Single Shell Commands

`M-!` (`shell-command`) reads a line of text using the minibuffer and executes it as a shell command, in a subshell made just for that command. Standard input for the command comes from the null device. If the shell command produces any output, the output appears either in the echo area (if it is short), or in an Emacs buffer named '`*Shell Command Output*`', displayed in another window (if the output is long).

For instance, one way to decompress a file named '`foo.gz`' is to type `M-! gunzip foo.gz RET`. That shell command normally creates the file '`foo`' and produces no terminal output.

A numeric argument to `shell-command`, e.g., `M-1 M-!`, causes it to insert terminal output into the current buffer instead of a separate buffer. It puts point before the output, and sets the mark after the output. For instance, `M-1 M-! gunzip < foo.gz RET` would insert the uncompressed form of the file '`foo.gz`' into the current buffer.

Provided the specified shell command does not end with '`&`', it runs *synchronously*, and you must wait for it to exit before continuing to use Emacs. To stop waiting, type `C-g` to quit; this sends a `SIGINT` signal to terminate the shell command (this is the same signal that `C-c` normally generates in the shell). Emacs then waits until the command actually terminates. If the shell command doesn't stop (because it ignores the `SIGINT` signal), type `C-g` again; this sends the command a `SIGKILL` signal, which is impossible to ignore.

A shell command that ends in '&' is executed *asynchronously*, and you can continue to use Emacs as it runs. You can also type M-& (async-shell-command) to execute a shell command asynchronously; this is exactly like calling M-! with a trailing '&', except that you do not need the '&'. The default output buffer for asynchronous shell commands is named '*Async Shell Command*'. Emacs inserts the output into this buffer as it comes in, whether or not the buffer is visible in a window.

If you want to run more than one asynchronous shell command at the same time, they could end up competing for the output buffer. The option async-shell-command-buffer specifies what to do about this; e.g., whether to rename the pre-existing output buffer, or to use a different buffer for the new command. Consult the variable's documentation for more possibilities.

M-| (shell-command-on-region) is like M-!, but passes the contents of the region as the standard input to the shell command, instead of no input. With a numeric argument, it deletes the old region and replaces it with the output from the shell command.

For example, you can use M-| with the gpg program to see what keys are in the buffer. If the buffer contains a GnuPG key, type C-x h M-| gpg RET to feed the entire buffer contents to gpg. This will output the list of keys to the '*Shell Command Output*' buffer.

The above commands use the shell specified by the variable shell-file-name. Its default value is determined by the SHELL environment variable when Emacs is started. If the file name is relative, Emacs searches the directories listed in exec-path (see Section 31.3 [Shell], page 655).

To specify a coding system for M-! or M-|, use the command C-x RET c immediately beforehand. See Section 19.11 [Communication Coding], page 387.

By default, error output is intermixed with the regular output in the output buffer. But if you change the value of the variable shell-command-default-error-buffer to a string, error output is inserted into a buffer of that name.

### 31.3.2 Interactive Subshell

To run a subshell interactively, type M-x shell. This creates (or reuses) a buffer named '*shell*', and runs a shell subprocess with input coming from and output going to that buffer. That is to say, any terminal output from the subshell goes into the buffer, advancing point, and any terminal input for the subshell comes from text in the buffer. To give input to the subshell, go to the end of the buffer and type the input, terminated by RET.

While the subshell is waiting or running a command, you can switch windows or buffers and perform other editing in Emacs. Emacs inserts the output from the subshell into the Shell buffer whenever it has time to process it (e.g., while waiting for keyboard input).

In the Shell buffer, prompts are displayed with the face comint-highlight-prompt, and submitted input lines are displayed with the face comint-highlight-input. This makes it easier to distinguish input lines from the shell output. See Section 11.12 [Faces], page 137.

To make multiple subshells, invoke M-x shell with a prefix argument (e.g., C-u M-x shell). Then the command will read a buffer name, and create (or reuse) a subshell in that buffer. You can also rename the '*shell*' buffer using M-x rename-uniquely, then create a new '*shell*' buffer using plain M-x shell. Subshells in different buffers run independently and in parallel.

To specify the shell file name used by `M-x shell`, customize the variable `explicit-shell-file-name`. If this is `nil` (the default), Emacs uses the environment variable `ESHELL` if it exists. Otherwise, it usually uses the variable `shell-file-name` (see Section 31.3.1 [Single Shell], page 655); but if the default directory is remote (see ⟨undefined⟩ [Remote Files], page ⟨undefined⟩), it prompts you for the shell file name.

Emacs sends the new shell the contents of the file '`~/.emacs_`*shellname*' as input, if it exists, where *shellname* is the name of the file that the shell was loaded from. For example, if you use bash, the file sent to it is '`~/.emacs_bash`'. If this file is not found, Emacs tries with '`~/.emacs.d/init_`*shellname*`.sh`'.

To specify a coding system for the shell, you can use the command `C-x RET c` immediately before `M-x shell`. You can also change the coding system for a running subshell by typing `C-x RET p` in the shell buffer. See Section 19.11 [Communication Coding], page 387.

Emacs sets the environment variable `INSIDE_EMACS` in the subshell to '*version*`,comint`', where *version* is the Emacs version (e.g., '`24.1`'). Programs can check this variable to determine whether they are running inside an Emacs subshell. (It also sets the `EMACS` environment variable to `t`, if that environment variable is not already defined. However, this environment variable is deprecated; programs that use it should switch to using `INSIDE_EMACS` instead.)

### 31.3.3 Shell Mode

The major mode for Shell buffers is Shell mode. Many of its special commands are bound to the `C-c` prefix, and resemble the usual editing and job control characters present in ordinary shells, except that you must type `C-c` first. Here is a list of Shell mode commands:

RET
: Send the current line as input to the subshell (`comint-send-input`). Any shell prompt at the beginning of the line is omitted (see Section 31.3.4 [Shell Prompts], page 659). If point is at the end of buffer, this is like submitting the command line in an ordinary interactive shell. However, you can also invoke RET elsewhere in the shell buffer to submit the current line as input.

TAB
: Complete the command name or file name before point in the shell buffer (`completion-at-point`). This uses the usual Emacs completion rules (see Section 5.4 [Completion], page 70), with the completion alternatives being file names, environment variable names, the shell command history, and history references (see Section 31.3.5.3 [History References], page 661).

  The variable `shell-completion-fignore` specifies a list of file name extensions to ignore in Shell mode completion. The default setting is `nil`, but some users prefer `("~" "#" "%")` to ignore file names ending in '`~`', '`#`' or '`%`'. Other related Comint modes use the variable `comint-completion-fignore` instead.

`M-?`
: Display temporarily a list of the possible completions of the file name before point (`comint-dynamic-list-filename-completions`).

`C-d`
: Either delete a character or send EOF (`comint-delchar-or-maybe-eof`). Typed at the end of the shell buffer, this sends EOF to the subshell. Typed at any other position in the buffer, this deletes a character as usual.

`C-c C-a`
: Move to the beginning of the line, but after the prompt if any (`comint-bol-or-process-mark`). If you repeat this command twice in a row, the second time it

moves back to the process mark, which is the beginning of the input that you have not yet sent to the subshell. (Normally that is the same place—the end of the prompt on this line—but after `C-c SPC` the process mark may be in a previous line.)

`C-c SPC` Accumulate multiple lines of input, then send them together. This command inserts a newline before point, but does not send the preceding text as input to the subshell—at least, not yet. Both lines, the one before this newline and the one after, will be sent together (along with the newline that separates them), when you type RET.

`C-c C-u` Kill all text pending at end of buffer to be sent as input (`comint-kill-input`). If point is not at end of buffer, this only kills the part of this text that precedes point.

`C-c C-w` Kill a word before point (`backward-kill-word`).

`C-c C-c` Interrupt the shell or its current subjob if any (`comint-interrupt-subjob`). This command also kills any shell input pending in the shell buffer and not yet sent.

`C-c C-z` Stop the shell or its current subjob if any (`comint-stop-subjob`). This command also kills any shell input pending in the shell buffer and not yet sent.

`C-c C-\` Send quit signal to the shell or its current subjob if any (`comint-quit-subjob`). This command also kills any shell input pending in the shell buffer and not yet sent.

`C-c C-o` Delete the last batch of output from a shell command (`comint-delete-output`). This is useful if a shell command spews out lots of output that just gets in the way.

`C-c C-s` Write the last batch of output from a shell command to a file (`comint-write-output`). With a prefix argument, the file is appended to instead. Any prompt at the end of the output is not written.

`C-c C-r`
`C-M-l` Scroll to display the beginning of the last batch of output at the top of the window; also move the cursor there (`comint-show-output`).

`C-c C-e` Scroll to put the end of the buffer at the bottom of the window (`comint-show-maximum-output`).

`C-c C-f` Move forward across one shell command, but not beyond the current line (`shell-forward-command`). The variable `shell-command-regexp` specifies how to recognize the end of a command.

`C-c C-b` Move backward across one shell command, but not beyond the current line (`shell-backward-command`).

`M-x dirs` Ask the shell for its working directory, and update the Shell buffer's default directory. See Section 31.3.6 [Directory Tracking], page 662.

`M-x send-invisible RET` *text* `RET`
Send *text* as input to the shell, after reading it without echoing. This is useful when a shell command runs a program that asks for a password.

Please note that Emacs will not echo passwords by default. If you really want them to be echoed, evaluate the following Lisp expression:

```
(remove-hook 'comint-output-filter-functions
             'comint-watch-for-password-prompt)
```

`M-x comint-continue-subjob`
: Continue the shell process. This is useful if you accidentally suspend the shell process.[3]

`M-x comint-strip-ctrl-m`
: Discard all control-M characters from the current group of shell output. The most convenient way to use this command is to make it run automatically when you get output from the subshell. To do that, evaluate this Lisp expression:

```
(add-hook 'comint-output-filter-functions
          'comint-strip-ctrl-m)
```

`M-x comint-truncate-buffer`
: This command truncates the shell buffer to a certain maximum number of lines, specified by the variable `comint-buffer-maximum-size`. Here's how to do this automatically each time you get output from the subshell:

```
(add-hook 'comint-output-filter-functions
          'comint-truncate-buffer)
```

Shell mode is a derivative of Comint mode, a general-purpose mode for communicating with interactive subprocesses. Most of the features of Shell mode actually come from Comint mode, as you can see from the command names listed above. The special features of Shell mode include the directory tracking feature, and a few user commands.

Other Emacs features that use variants of Comint mode include GUD (see Section 24.6 [Debuggers], page 539) and `M-x run-lisp` (see Section 24.11 [External Lisp], page 552).

You can use `M-x comint-run` to execute any program of your choice in a subprocess using unmodified Comint mode—without the specializations of Shell mode.

### 31.3.4 Shell Prompts

A prompt is text output by a program to show that it is ready to accept new user input. Normally, Comint mode (and thus Shell mode) automatically figures out part of the buffer is a prompt, based on the output of the subprocess. (Specifically, it assumes that any received output line which doesn't end with a newline is a prompt.)

Comint mode divides the buffer into two types of *fields*: input fields (where user input is typed) and output fields (everywhere else). Prompts are part of the output fields. Most Emacs motion commands do not cross field boundaries, unless they move over multiple lines. For instance, when point is in the input field on a shell command line, `C-a` puts point at the beginning of the input field, after the prompt. Internally, the fields are implemented using the `field` text property (see Section "Text Properties" in *the Emacs Lisp Reference Manual*).

[3] You should not suspend the shell process. Suspending a subjob of the shell is a completely different matter—that is normal practice, but you must use the shell to continue the subjob; this command won't do it.

If you change the variable `comint-use-prompt-regexp` to a non-`nil` value, then Comint mode recognize prompts using a regular expression (see Section 12.6 [Regexps], page 206). In Shell mode, the regular expression is specified by the variable `shell-prompt-pattern`. The default value of `comint-use-prompt-regexp` is `nil`, because this method for recognizing prompts is unreliable, but you may want to set it to a non-`nil` value in unusual circumstances. In that case, Emacs does not divide the Comint buffer into fields, so the general motion commands behave as they normally do in buffers without special text properties. However, you can use the paragraph motion commands to conveniently navigate the buffer (see ⟨undefined⟩ [Paragraphs], page ⟨undefined⟩); in Shell mode, Emacs uses `shell-prompt-pattern` as paragraph boundaries.

### 31.3.5 Shell Command History

Shell buffers support three ways of repeating earlier commands. You can use keys like those used for the minibuffer history; these work much as they do in the minibuffer, inserting text from prior commands while point remains always at the end of the buffer. You can move through the buffer to previous inputs in their original place, then resubmit them or copy them to the end. Or you can use a '`!`'-style history reference.

#### 31.3.5.1 Shell History Ring

`M-p`
`C-UP` Fetch the next earlier old shell command.

`M-n`
`C-DOWN` Fetch the next later old shell command.

`M-r` Begin an incremental regexp search of old shell commands.

`C-c C-x` Fetch the next subsequent command from the history.

`C-c .` Fetch one argument from an old shell command.

`C-c C-l` Display the buffer's history of shell commands in another window (`comint-dynamic-list-input-ring`).

Shell buffers provide a history of previously entered shell commands. To reuse shell commands from the history, use the editing commands `M-p`, `M-n`, `M-r` and `M-s`. These work just like the minibuffer history commands (see Section 5.5 [Minibuffer History], page 74), except that they operate within the Shell buffer rather than the minibuffer.

`M-p` fetches an earlier shell command to the end of the shell buffer. Successive use of `M-p` fetches successively earlier shell commands, each replacing any text that was already present as potential shell input. `M-n` does likewise except that it finds successively more recent shell commands from the buffer. `C-UP` works like `M-p`, and `C-DOWN` like `M-n`.

The history search command `M-r` begins an incremental regular expression search of previous shell commands. After typing `M-r`, start typing the desired string or regular expression; the last matching shell command will be displayed in the current line. Incremental search commands have their usual effects—for instance, `C-s` and `C-r` search forward and backward for the next match (see Section 12.1 [Incremental Search], page 199). When you find the desired input, type RET to terminate the search. This puts the input in the command line. Any partial input you were composing before navigating the history list is restored when you go to the beginning or end of the history ring.

Often it is useful to reexecute several successive shell commands that were previously executed in sequence. To do this, first find and reexecute the first command of the sequence. Then type `C-c C-x`; that will fetch the following command—the one that follows the command you just repeated. Then type RET to reexecute this command. You can reexecute several successive commands by typing `C-c C-x RET` over and over.

The command `C-c .` (`comint-input-previous-argument`) copies an individual argument from a previous command, like `ESC .` in Bash. The simplest use copies the last argument from the previous shell command. With a prefix argument *n*, it copies the *n*th argument instead. Repeating `C-c .` copies from an earlier shell command instead, always using the same value of *n* (don't give a prefix argument when you repeat the `C-c .` command).

These commands get the text of previous shell commands from a special history list, not from the shell buffer itself. Thus, editing the shell buffer, or even killing large parts of it, does not affect the history that these commands access.

Some shells store their command histories in files so that you can refer to commands from previous shell sessions. Emacs reads the command history file for your chosen shell, to initialize its own command history. The file name is '`~/.bash_history`' for bash, '`~/.sh_history`' for ksh, and '`~/.history`' for other shells.

### 31.3.5.2 Shell History Copying

`C-c C-p` Move point to the previous prompt (`comint-previous-prompt`).

`C-c C-n` Move point to the following prompt (`comint-next-prompt`).

`C-c RET` Copy the input command at point, inserting the copy at the end of the buffer (`comint-copy-old-input`). This is useful if you move point back to a previous command. After you copy the command, you can submit the copy as input with RET. If you wish, you can edit the copy before resubmitting it. If you use this command on an output line, it copies that line to the end of the buffer.

`Mouse-2` If `comint-use-prompt-regexp` is `nil` (the default), copy the old input command that you click on, inserting the copy at the end of the buffer (`comint-insert-input`). If `comint-use-prompt-regexp` is non-`nil`, or if the click is not over old input, just yank as usual.

Moving to a previous input and then copying it with `C-c RET` or `Mouse-2` produces the same results—the same buffer contents—that you would get by using `M-p` enough times to fetch that previous input from the history list. However, `C-c RET` copies the text from the buffer, which can be different from what is in the history list if you edit the input text in the buffer after it has been sent.

### 31.3.5.3 Shell History References

Various shells including csh and bash support *history references* that begin with '`!`' and '`^`'. Shell mode recognizes these constructs, and can perform the history substitution for you.

If you insert a history reference and type TAB, this searches the input history for a matching command, performs substitution if necessary, and places the result in the buffer in place of the history reference. For example, you can fetch the most recent command

beginning with 'mv' with ! m v TAB. You can edit the command if you wish, and then resubmit the command to the shell by typing RET.

Shell mode can optionally expand history references in the buffer when you send them to the shell. To request this, set the variable comint-input-autoexpand to input. You can make SPC perform history expansion by binding SPC to the command comint-magic-space.

Shell mode recognizes history references when they follow a prompt. See Section 31.3.4 [Shell Prompts], page 659, for how Shell mode recognizes prompts.

### 31.3.6 Directory Tracking

Shell mode keeps track of 'cd', 'pushd' and 'popd' commands given to the subshell, in order to keep the Shell buffer's default directory (see Section 15.8 [File Names], page 251) the same as the shell's working directory. It recognizes these commands by examining lines of input that you send.

If you use aliases for these commands, you can tell Emacs to recognize them also, by setting the variables shell-pushd-regexp, shell-popd-regexp, and shell-cd-regexp to the appropriate regular expressions (see Section 12.6 [Regexps], page 206). For example, if shell-pushd-regexp matches the beginning of a shell command line, that line is regarded as a pushd command. These commands are recognized only at the beginning of a shell command line.

If Emacs gets confused about changes in the working directory of the subshell, type M-x dirs. This command asks the shell for its working directory and updates the default directory accordingly. It works for shells that support the most common command syntax, but may not work for unusual shells.

You can also use Dirtrack mode, a buffer-local minor mode that implements an alternative method of tracking the shell's working directory. To use this method, your shell prompt must contain the working directory at all times, and you must supply a regular expression for recognizing which part of the prompt contains the working directory; see the documentation of the variable dirtrack-list for details. To use Dirtrack mode, type M-x dirtrack-mode in the Shell buffer, or add dirtrack-mode to shell-mode-hook (see Section 33.2.2 [Hooks], page 696).

### 31.3.7 Shell Mode Options

If the variable comint-scroll-to-bottom-on-input is non-nil, insertion and yank commands scroll the selected window to the bottom before inserting. The default is nil.

If comint-scroll-show-maximum-output is non-nil, then arrival of output when point is at the end tries to scroll the last line of text to the bottom line of the window, showing as much useful text as possible. (This mimics the scrolling behavior of most terminals.) The default is t.

By setting comint-move-point-for-output, you can opt for having point jump to the end of the buffer whenever output arrives—no matter where in the buffer point was before. If the value is this, point jumps in the selected window. If the value is all, point jumps in each window that shows the Comint buffer. If the value is other, point jumps in all nonselected windows that show the current buffer. The default value is nil, which means point does not jump to the end.

If you set `comint-prompt-read-only`, the prompts in the Comint buffer are read-only.

The variable `comint-input-ignoredups` controls whether successive identical inputs are stored in the input history. A non-`nil` value means to omit an input that is the same as the previous input. The default is `nil`, which means to store each input even if it is equal to the previous input.

Three variables customize file name completion. The variable `comint-completion-addsuffix` controls whether completion inserts a space or a slash to indicate a fully completed file or directory name (non-`nil` means do insert a space or slash). `comint-completion-recexact`, if non-`nil`, directs TAB to choose the shortest possible completion if the usual Emacs completion algorithm cannot add even a single character. `comint-completion-autolist`, if non-`nil`, says to list all the possible completions whenever completion is not exact.

Command completion normally considers only executable files. If you set `shell-completion-execonly` to `nil`, it considers nonexecutable files as well.

You can configure the behavior of '`pushd`'. Variables control whether '`pushd`' behaves like '`cd`' if no argument is given (`shell-pushd-tohome`), pop rather than rotate with a numeric argument (`shell-pushd-dextract`), and only add directories to the directory stack if they are not already on it (`shell-pushd-dunique`). The values you choose should match the underlying shell, of course.

### 31.3.8 Emacs Terminal Emulator

To run a subshell in a text terminal emulator, use `M-x term`. This creates (or reuses) a buffer named '`*terminal*`', and runs a subshell with input coming from your keyboard, and output going to that buffer.

The terminal emulator uses Term mode, which has two input modes. In *line mode*, Term basically acts like Shell mode (see Section 31.3.3 [Shell Mode], page 657). In *char mode*, each character is sent directly to the subshell, as terminal input; the sole exception is the terminal escape character, which by default is `C-c` (see Section 31.3.9 [Term Mode], page 664). Any echoing of your input is the responsibility of the subshell; any terminal output from the subshell goes into the buffer, advancing point.

Some programs (such as Emacs itself) need to control the appearance of the terminal screen in detail. They do this by emitting special control codes. Term mode recognizes and handles ANSI-standard VT100-style escape sequences, which are accepted by most modern terminals, including `xterm`. (Hence, you can actually run Emacs inside an Emacs Term window.)

The `term` face specifies the default appearance of text in the terminal emulator (the default is the same appearance as the `default` face). When terminal control codes are used to change the appearance of text, these are represented in the terminal emulator by the faces `term-color-black`, `term-color-red`, `term-color-green`, `term-color-yellow` `term-color-blue`, `term-color-magenta`, `term-color-cyan`, `term-color-white`, `term-color-underline`, and `term-color-bold`. See Section 11.12 [Faces], page 137.

You can also Term mode to communicate with a device connected to a serial port. See Section 31.3.11 [Serial Terminal], page 665.

The file name used to load the subshell is determined the same way as for Shell mode. To make multiple terminal emulators, rename the buffer '`*terminal*`' to something different using `M-x rename-uniquely`, just as with Shell mode.

Unlike Shell mode, Term mode does not track the current directory by examining your input. But some shells can tell Term what the current directory is. This is done automatically by `bash` version 1.15 and later.

### 31.3.9 Term Mode

The terminal emulator uses Term mode, which has two input modes. In line mode, Term basically acts like Shell mode (see Section 31.3.3 [Shell Mode], page 657). In char mode, each character is sent directly to the subshell, except for the Term escape character, normally `C-c`.

To switch between line and char mode, use these commands:

`C-c C-j` Switch to line mode (`term-line-mode`). Do nothing if already in line mode.

`C-c C-k` Switch to char mode (`term-char-mode`). Do nothing if already in char mode.

The following commands are only available in char mode:

`C-c C-c` Send a literal C-C to the sub-shell.

`C-c` *char* This is equivalent to `C-x` *char* in normal Emacs. For example, `C-c o` invokes the global binding of `C-x o`, which is normally '`other-window`'.

Term mode has a page-at-a-time feature. When enabled, it makes output pause at the end of each screenful:

`C-c C-q` Toggle the page-at-a-time feature. This command works in both line and char modes. When the feature is enabled, the mode-line displays the word '`page`', and each time Term receives more than a screenful of output, it pauses and displays '`**MORE**`' in the mode-line. Type SPC to display the next screenful of output, or `?` to see your other options. The interface is similar to the `more` program.

### 31.3.10 Remote Host Shell

You can login to a remote computer, using whatever commands you would from a regular terminal (e.g., using the `telnet` or `rlogin` commands), from a Term window.

A program that asks you for a password will normally suppress echoing of the password, so the password will not show up in the buffer. This will happen just as if you were using a real terminal, if the buffer is in char mode. If it is in line mode, the password is temporarily visible, but will be erased when you hit return. (This happens automatically; there is no special password processing.)

When you log in to a different machine, you need to specify the type of terminal you're using, by setting the `TERM` environment variable in the environment for the remote login command. (If you use bash, you do that by writing the variable assignment before the remote login command, without a separating comma.) Terminal types '`ansi`' or '`vt100`' will work on most systems.

### 31.3.11 Serial Terminal

If you have a device connected to a serial port of your computer, you can communicate with it by typing `M-x serial-term`. This command asks for a serial port name and speed, and switches to a new Term mode buffer. Emacs communicates with the serial device through this buffer just like it does with a terminal in ordinary Term mode.

The speed of the serial port is measured in bits per second. The most common speed is 9600 bits per second. You can change the speed interactively by clicking on the mode line.

A serial port can be configured even more by clicking on "8N1" in the mode line. By default, a serial port is configured as "8N1", which means that each byte consists of 8 data bits, No parity check bit, and 1 stopbit.

If the speed or the configuration is wrong, you cannot communicate with your device and will probably only see garbage output in the window.

## 31.4 Using Emacs as a Server

Various programs can invoke your choice of editor to edit a particular piece of text. For instance, version control programs invoke an editor to enter version control logs (see Section 25.1 [Version Control], page 554), and the Unix `mail` utility invokes an editor to enter a message to send. By convention, your choice of editor is specified by the environment variable `EDITOR`. If you set `EDITOR` to '`emacs`', Emacs would be invoked, but in an inconvenient way—by starting a new Emacs process. This is inconvenient because the new Emacs process doesn't share buffers, a command history, or other kinds of information with any existing Emacs process.

You can solve this problem by setting up Emacs as an *edit server*, so that it "listens" for external edit requests and acts accordingly. There are two ways to start an Emacs server:

- Run the command `server-start` in an existing Emacs process: either type `M-x server-start`, or put the expression `(server-start)` in your init file (see Section 33.4 [Init File], page 711). The existing Emacs process is the server; when you exit Emacs, the server dies with the Emacs process.
- Run Emacs as a *daemon*, using the '`--daemon`' command-line option. See Section E.2 [Initial Options], page 755. When Emacs is started this way, it calls `server-start` after initialization, and returns control to the calling terminal instead of opening an initial frame; it then waits in the background, listening for edit requests.

Either way, once an Emacs server is started, you can use a shell command called `emacsclient` to connect to the Emacs process and tell it to visit a file. You can then set the `EDITOR` environment variable to '`emacsclient`', so that external programs will use the existing Emacs process for editing.[4]

You can run multiple Emacs servers on the same machine by giving each one a unique "server name", using the variable `server-name`. For example, `M-x set-variable RET server-name RET foo RET` sets the server name to '`foo`'. The `emacsclient` program can specify a server by name, using the '`-s`' option (see Section 31.4.2 [emacsclient Options], page 667).

[4] Some programs use a different environment variable; for example, to make TEX use '`emacsclient`', set the `TEXEDIT` environment variable to '`emacsclient +%d %s`'.

If you have defined a server by a unique server name, it is possible to connect to the server from another Emacs instance and evaluate Lisp expressions on the server, using the `server-eval-at` function. For instance, `(server-eval-at "foo" '(+ 1 2))` evaluates the expression `(+ 1 2)` on the '`foo`' server, and returns `3`. (If there is no server with that name, an error is signaled.) Currently, this feature is mainly useful for developers.

### 31.4.1 Invoking `emacsclient`

The simplest way to use the `emacsclient` program is to run the shell command '`emacsclient` *`file`*', where *file* is a file name. This connects to an Emacs server, and tells that Emacs process to visit *file* in one of its existing frames—either a graphical frame, or one in a text terminal (see Chapter 18 [Frames], page 341). You can then select that frame to begin editing.

If there is no Emacs server, the `emacsclient` program halts with an error message. If the Emacs process has no existing frame—which can happen if it was started as a daemon (see Section 31.4 [Emacs Server], page 665)—then Emacs opens a frame on the terminal in which you called `emacsclient`.

You can also force `emacsclient` to open a new frame on a graphical display, or on a text terminal, using the '`-c`' and '`-t`' options. See Section 31.4.2 [emacsclient Options], page 667.

If you are running on a single text terminal, you can switch between `emacsclient`'s shell and the Emacs server using one of two methods: (i) run the Emacs server and `emacsclient` on different virtual terminals, and switch to the Emacs server's virtual terminal after calling `emacsclient`; or (ii) call `emacsclient` from within the Emacs server itself, using Shell mode (see Section 31.3.2 [Interactive Shell], page 656) or Term mode (see Section 31.3.9 [Term Mode], page 664); `emacsclient` blocks only the subshell under Emacs, and you can still use Emacs to edit the file.

When you finish editing *file* in the Emacs server, type `C-x #` (`server-edit`) in its buffer. This saves the file and sends a message back to the `emacsclient` program, telling it to exit. Programs that use `EDITOR` usually wait for the "editor"—in this case `emacsclient`—to exit before doing something else.

You can also call `emacsclient` with multiple file name arguments: '`emacsclient` *`file1 file2 ...`*' tells the Emacs server to visit *file1*, *file2*, and so forth. Emacs selects the buffer visiting *file1*, and buries the other buffers at the bottom of the buffer list (see Chapter 16 [Buffers], page 272). The `emacsclient` program exits once all the specified files are finished (i.e., once you have typed `C-x #` in each server buffer).

Finishing with a server buffer also kills the buffer, unless it already existed in the Emacs session before the server was asked to create it. However, if you set `server-kill-new-buffers` to `nil`, then a different criterion is used: finishing with a server buffer kills it if the file name matches the regular expression `server-temp-file-regexp`. This is set up to distinguish certain "temporary" files.

Each `C-x #` checks for other pending external requests to edit various files, and selects the next such file. You can switch to a server buffer manually if you wish; you don't have to arrive at it with `C-x #`. But `C-x #` is the way to tell `emacsclient` that you are finished.

If you set the value of the variable `server-window` to a window or a frame, `C-x #` always displays the next server buffer in that window or in that frame.

### 31.4.2 `emacsclient` Options

You can pass some optional arguments to the `emacsclient` program, such as:

```
emacsclient -c +12 file1 +4:3 file2
```

The '`+line`' or '`+line:column`' arguments specify line numbers, or line and column numbers, for the next file argument. These behave like the command line arguments for Emacs itself. See Section E.1 [Action Arguments], page 754.

The other optional arguments recognized by `emacsclient` are listed below:

'`-a command`'
'`--alternate-editor=command`'
: Specify a command to run if `emacsclient` fails to contact Emacs. This is useful when running `emacsclient` in a script.

  As a special exception, if *command* is the empty string, then `emacsclient` starts Emacs in daemon mode (as `emacs --daemon`) and then tries connecting again.

  The environment variable `ALTERNATE_EDITOR` has the same effect as the '`-a`' option. If both are present, the latter takes precedence.

'`-c`'
: Create a new graphical *client frame*, instead of using an existing Emacs frame. See below for the special behavior of `C-x C-c` in a client frame. If Emacs cannot create a new graphical frame (e.g., if it cannot connect to the X server), it tries to create a text terminal client frame, as though you had supplied the '`-t`' option instead.

  On MS-Windows, a single Emacs session cannot display frames on both graphical and text terminals, nor on multiple text terminals. Thus, if the Emacs server is running on a text terminal, the '`-c`' option, like the '`-t`' option, creates a new frame in the server's current text terminal. See Section I.1 [Windows Startup], page 781.

  If you omit a filename argument while supplying the '`-c`' option, the new frame displays the '`*scratch*`' buffer by default. If `initial-buffer-choice` is a string (see Section 3.1 [Entering Emacs], page 56), the new frame displays that file or directory instead.

'`-F alist`'
'`--frame-parameters=alist`'
: Set the parameters for a newly-created graphical frame (see Section 18.3 [Frame Parameters], page 345).

'`-d display`'
'`--display=display`'
: Tell Emacs to open the given files on the X display *display* (assuming there is more than one X display available).

'`-e`'
'`--eval`'
: Tell Emacs to evaluate some Emacs Lisp code, instead of visiting some files. When this option is given, the arguments to `emacsclient` are interpreted as a list of expressions to evaluate, *not* as a list of files to visit.

‘`-f server-file`’
‘`--server-file=server-file`’
: Specify a *server file* for connecting to an Emacs server via TCP.

  An Emacs server usually uses an operating system feature called a “local socket” to listen for connections. Some operating systems, such as Microsoft Windows, do not support local sockets; in that case, the server communicates with `emacsclient` via TCP.

  When you start a TCP Emacs server, Emacs creates a *server file* containing the TCP information to be used by `emacsclient` to connect to the server. The variable `server-auth-dir` specifies the directory containing the server file; by default, this is ‘`~/.emacs.d/server/`’. To tell `emacsclient` to connect to the server over TCP with a specific server file, use the ‘`-f`’ or ‘`--server-file`’ option, or set the `EMACS_SERVER_FILE` environment variable.

‘`-n`’
‘`--no-wait`’
: Let `emacsclient` exit immediately, instead of waiting until all server buffers are finished. You can take as long as you like to edit the server buffers within Emacs, and they are *not* killed when you type `C-x #` in them.

‘`--parent-id ID`’
: Open an `emacsclient` frame as a client frame in the parent X window with id *ID*, via the XEmbed protocol. Currently, this option is mainly useful for developers.

‘`-q`’
‘`--quiet`’
: Do not let `emacsclient` display messages about waiting for Emacs or connecting to remote server sockets.

‘`-s server-name`’
‘`--socket-name=server-name`’
: Connect to the Emacs server named *server-name*. The server name is given by the variable `server-name` on the Emacs server. If this option is omitted, `emacsclient` connects to the first server it finds. (This option is not supported on MS-Windows.)

‘`-t`’
‘`--tty`’
‘`-nw`’
: Create a new client frame on the current text terminal, instead of using an existing Emacs frame. This behaves just like the ‘`-c`’ option, described above, except that it creates a text terminal frame (see ⟨undefined⟩ [Non-Window Terminals], page ⟨undefined⟩).

  On MS-Windows, ‘`-t`’ behaves just like ‘`-c`’ if the Emacs server is using the graphical display, but if the Emacs server is running on a text terminal, it creates a new frame in the current text terminal.

The new graphical or text terminal frames created by the ‘`-c`’ or ‘`-t`’ options are considered *client frames*. Any new frame that you create from a client frame is also considered a client frame. If you type `C-x C-c` (`save-buffers-kill-terminal`) in a client frame, that command does not kill the Emacs session as it normally does (see Section 3.2 [Exiting],

page 57). Instead, Emacs deletes the client frame; furthermore, if the client frame has an `emacsclient` waiting to regain control (i.e., if you did not supply the '`-n`' option), Emacs deletes all other frames of the same client, and marks the client's server buffers as finished, as though you had typed `C-x #` in all of them. If it so happens that there are no remaining frames after the client frame(s) are deleted, the Emacs session exits.

As an exception, when Emacs is started as a daemon, all frames are considered client frames, and `C-x C-c` never kills Emacs. To kill a daemon session, type `M-x kill-emacs`.

Note that the '`-t`' and '`-n`' options are contradictory: '`-t`' says to take control of the current text terminal to create a new client frame, while '`-n`' says not to take control of the text terminal. If you supply both options, Emacs visits the specified files(s) in an existing frame rather than a new client frame, negating the effect of '`-t`'.

## 31.5 Printing Hard Copies

Emacs provides commands for printing hardcopies of either an entire buffer or part of one. You can invoke the printing commands directly, as detailed below, or using the '`File`' menu on the menu bar.

Aside from the commands described in this section, you can also print hardcopies from Dired (see Section 27.7 [Operating on Files], page 593) and the diary (see Section 28.10.1 [Displaying the Diary], page 615). You can also "print" an Emacs buffer to HTML with the command `M-x htmlfontify-buffer`, which converts the current buffer to a HTML file, replacing Emacs faces with CSS-based markup. Furthermore, Org mode allows you to "print" Org files to a variety of formats, such as PDF (see ⟨undefined⟩ [Org Mode], page ⟨undefined⟩).

`M-x print-buffer`
: Print hardcopy of current buffer with page headings containing the file name and page number.

`M-x lpr-buffer`
: Print hardcopy of current buffer without page headings.

`M-x print-region`
: Like `print-buffer` but print only the current region.

`M-x lpr-region`
: Like `lpr-buffer` but print only the current region.

On most operating system, the above hardcopy commands submit files for printing by calling the `lpr` program. To change the printer program, customize the variable `lpr-command`. To specify extra switches to give the printer program, customize the list variable `lpr-switches`. Its value should be a list of option strings, each of which should start with '`-`' (e.g., the option string `"-w80"` specifies a line width of 80 columns). The default is the empty list, `nil`.

To specify the printer to use, set the variable `printer-name`. The default, `nil`, specifies the default printer. If you set it to a printer name (a string), that name is passed to `lpr` with the '`-P`' switch; if you are not using `lpr`, you should specify the switch with `lpr-printer-switch`.

The variable `lpr-headers-switches` similarly specifies the extra switches to use to make page headers. The variable `lpr-add-switches` controls whether to supply '`-T`' and '`-J`' options (suitable for `lpr`) to the printer program: `nil` means don't add them (this should be the value if your printer program is not compatible with `lpr`).

### 31.5.1 PostScript Hardcopy

These commands convert buffer contents to PostScript, either printing it or leaving it in another Emacs buffer.

`M-x ps-print-buffer`
: Print hardcopy of the current buffer in PostScript form.

`M-x ps-print-region`
: Print hardcopy of the current region in PostScript form.

`M-x ps-print-buffer-with-faces`
: Print hardcopy of the current buffer in PostScript form, showing the faces used in the text by means of PostScript features.

`M-x ps-print-region-with-faces`
: Print hardcopy of the current region in PostScript form, showing the faces used in the text.

`M-x ps-spool-buffer`
: Generate and spool a PostScript image for the current buffer text.

`M-x ps-spool-region`
: Generate and spool a PostScript image for the current region.

`M-x ps-spool-buffer-with-faces`
: Generate and spool a PostScript image for the current buffer, showing the faces used.

`M-x ps-spool-region-with-faces`
: Generate and spool a PostScript image for the current region, showing the faces used.

`M-x ps-despool`
: Send the spooled PostScript to the printer.

`M-x handwrite`
: Generate/print PostScript for the current buffer as if handwritten.

The `ps-print-buffer` and `ps-print-region` commands print buffer contents in PostScript form. One command prints the entire buffer; the other, just the region. The commands `ps-print-buffer-with-faces` and `ps-print-region-with-faces` behave similarly, but use PostScript features to show the faces (fonts and colors) of the buffer text.

Interactively, when you use a prefix argument (`C-u`), the command prompts the user for a file name, and saves the PostScript image in that file instead of sending it to the printer.

The commands whose names have '`spool`' instead of '`print`', generate the PostScript output in an Emacs buffer instead of sending it to the printer.

Use the command `ps-despool` to send the spooled images to the printer. This command sends the PostScript generated by '`-spool-`' commands (see commands above) to

the printer. With a prefix argument (`C-u`), it prompts for a file name, and saves the spooled PostScript image in that file instead of sending it to the printer.

`M-x handwrite` is more frivolous. It generates a PostScript rendition of the current buffer as a cursive handwritten document. It can be customized in group `handwrite`. This function only supports ISO 8859-1 characters.

### 31.5.2 Variables for PostScript Hardcopy

All the PostScript hardcopy commands use the variables `ps-lpr-command` and `ps-lpr-switches` to specify how to print the output. `ps-lpr-command` specifies the command name to run, `ps-lpr-switches` specifies command line options to use, and `ps-printer-name` specifies the printer. If you don't set the first two variables yourself, they take their initial values from `lpr-command` and `lpr-switches`. If `ps-printer-name` is `nil`, `printer-name` is used.

The variable `ps-print-header` controls whether these commands add header lines to each page—set it to `nil` to turn headers off.

If your printer doesn't support colors, you should turn off color processing by setting `ps-print-color-p` to `nil`. By default, if the display supports colors, Emacs produces hardcopy output with color information; on black-and-white printers, colors are emulated with shades of gray. This might produce illegible output, even if your screen colors only use shades of gray.

Alternatively, you can set `ps-print-color-p` to `black-white` to print colors on black/white printers.

By default, PostScript printing ignores the background colors of the faces, unless the variable `ps-use-face-background` is non-`nil`. This is to avoid unwanted interference with the zebra stripes and background image/text.

The variable `ps-paper-type` specifies which size of paper to format for; legitimate values include `a4`, `a3`, `a4small`, `b4`, `b5`, `executive`, `ledger`, `legal`, `letter`, `letter-small`, `statement`, `tabloid`. The default is `letter`. You can define additional paper sizes by changing the variable `ps-page-dimensions-database`.

The variable `ps-landscape-mode` specifies the orientation of printing on the page. The default is `nil`, which stands for "portrait" mode. Any non-`nil` value specifies "landscape" mode.

The variable `ps-number-of-columns` specifies the number of columns; it takes effect in both landscape and portrait mode. The default is 1.

The variable `ps-font-family` specifies which font family to use for printing ordinary text. Legitimate values include `Courier`, `Helvetica`, `NewCenturySchlbk`, `Palatino` and `Times`. The variable `ps-font-size` specifies the size of the font for ordinary text. It defaults to 8.5 points.

Emacs supports more scripts and characters than a typical PostScript printer. Thus, some of the characters in your buffer might not be printable using the fonts built into your printer. You can augment the fonts supplied with the printer with those from the GNU Intlfonts package, or you can instruct Emacs to use Intlfonts exclusively. The variable `ps-multibyte-buffer` controls this: the default value, `nil`, is appropriate for printing ASCII and Latin-1 characters; a value of `non-latin-printer` is for printers which have the fonts

for ASCII, Latin-1, Japanese, and Korean characters built into them. A value of `bdf-font` arranges for the BDF fonts from the Intlfonts package to be used for *all* characters. Finally, a value of `bdf-font-except-latin` instructs the printer to use built-in fonts for ASCII and Latin-1 characters, and Intlfonts BDF fonts for the rest.

To be able to use the BDF fonts, Emacs needs to know where to find them. The variable `bdf-directory-list` holds the list of directories where Emacs should look for the fonts; the default value includes a single directory '`/usr/local/share/emacs/fonts/bdf`'.

Many other customization variables for these commands are defined and described in the Lisp files '`ps-print.el`' and '`ps-mule.el`'.

### 31.5.3 Printing Package

The basic Emacs facilities for printing hardcopy can be extended using the Printing package. This provides an easy-to-use interface for choosing what to print, previewing PostScript files before printing, and setting various printing options such as print headers, landscape or portrait modes, duplex modes, and so forth. On GNU/Linux or Unix systems, the Printing package relies on the '`gs`' and '`gv`' utilities, which are distributed as part of the GhostScript program. On MS-Windows, the '`gstools`' port of Ghostscript can be used.

To use the Printing package, add `(require 'printing)` to your init file (see Section 33.4 [Init File], page 711), followed by `(pr-update-menus)`. This function replaces the usual printing commands in the menu bar with a '`Printing`' submenu that contains various printing options. You can also type `M-x pr-interface RET`; this creates a '`*Printing Interface*`' buffer, similar to a customization buffer, where you can set the printing options. After selecting what and how to print, you start the print job using the '`Print`' button (click `mouse-2` on it, or move point over it and type `RET`). For further information on the various options, use the '`Interface Help`' button.

## 31.6 Sorting Text

Emacs provides several commands for sorting text in the buffer. All operate on the contents of the region. They divide the text of the region into many *sort records*, identify a *sort key* for each record, and then reorder the records into the order determined by the sort keys. The records are ordered so that their keys are in alphabetical order, or, for numeric sorting, in numeric order. In alphabetic sorting, all upper-case letters 'A' through 'Z' come before lower-case 'a', in accord with the ASCII character sequence.

The various sort commands differ in how they divide the text into sort records and in which part of each record is used as the sort key. Most of the commands make each line a separate sort record, but some commands use paragraphs or pages as sort records. Most of the sort commands use each entire sort record as its own sort key, but some use only a portion of the record as the sort key.

`M-x sort-lines`
: Divide the region into lines, and sort by comparing the entire text of a line. A numeric argument means sort into descending order.

`M-x sort-paragraphs`
: Divide the region into paragraphs, and sort by comparing the entire text of a paragraph (except for leading blank lines). A numeric argument means sort into descending order.

`M-x sort-pages`
: Divide the region into pages, and sort by comparing the entire text of a page (except for leading blank lines). A numeric argument means sort into descending order.

`M-x sort-fields`
: Divide the region into lines, and sort by comparing the contents of one field in each line. Fields are defined as separated by whitespace, so the first run of consecutive non-whitespace characters in a line constitutes field 1, the second such run constitutes field 2, etc.

  Specify which field to sort by with a numeric argument: 1 to sort by field 1, etc. A negative argument means count fields from the right instead of from the left; thus, minus 1 means sort by the last field. If several lines have identical contents in the field being sorted, they keep the same relative order that they had in the original buffer.

`M-x sort-numeric-fields`
: Like `M-x sort-fields` except the specified field is converted to an integer for each line, and the numbers are compared. '`10`' comes before '`2`' when considered as text, but after it when considered as a number. By default, numbers are interpreted according to `sort-numeric-base`, but numbers beginning with '`0x`' or '`0`' are interpreted as hexadecimal and octal, respectively.

`M-x sort-columns`
: Like `M-x sort-fields` except that the text within each line used for comparison comes from a fixed range of columns. See below for an explanation.

`M-x reverse-region`
: Reverse the order of the lines in the region. This is useful for sorting into descending order by fields or columns, since those sort commands do not have a feature for doing that.

For example, if the buffer contains this:

```
On systems where clash detection (locking of files being edited) is
implemented, Emacs also checks the first time you modify a buffer
whether the file has changed on disk since it was last visited or
saved.  If it has, you are asked to confirm that you want to change
the buffer.
```

applying `M-x sort-lines` to the entire buffer produces this:

```
On systems where clash detection (locking of files being edited) is
implemented, Emacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
whether the file has changed on disk since it was last visited or
```

where the upper-case '`O`' sorts before all lower-case letters. If you use `C-u 2 M-x sort-fields` instead, you get this:

```
implemented, Emacs also checks the first time you modify a buffer
saved.  If it has, you are asked to confirm that you want to change
the buffer.
On systems where clash detection (locking of files being edited) is
whether the file has changed on disk since it was last visited or
```

where the sort keys were ‘Emacs’, ‘If’, ‘buffer’, ‘systems’ and ‘the’.

M-x sort-columns requires more explanation. You specify the columns by putting point at one of the columns and the mark at the other column. Because this means you cannot put point or the mark at the beginning of the first line of the text you want to sort, this command uses an unusual definition of “region”: all of the line point is in is considered part of the region, and so is all of the line the mark is in, as well as all the lines in between.

For example, to sort a table by information found in columns 10 to 15, you could put the mark on column 10 in the first line of the table, and point on column 15 in the last line of the table, and then run sort-columns. Equivalently, you could run it with the mark on column 15 in the first line and point on column 10 in the last line.

This can be thought of as sorting the rectangle specified by point and the mark, except that the text on each line to the left or right of the rectangle moves along with the text inside the rectangle. See Section 9.5 [Rectangles], page 103.

Many of the sort commands ignore case differences when comparing, if sort-fold-case is non-nil.

## 31.7 Editing Binary Files

There is a special major mode for editing binary files: Hexl mode. To use it, use M-x hexl-find-file instead of C-x C-f to visit the file. This command converts the file’s contents to hexadecimal and lets you edit the translation. When you save the file, it is converted automatically back to binary.

You can also use M-x hexl-mode to translate an existing buffer into hex. This is useful if you visit a file normally and then discover it is a binary file.

Ordinary text characters overwrite in Hexl mode. This is to reduce the risk of accidentally spoiling the alignment of data in the file. There are special commands for insertion. Here is a list of the commands of Hexl mode:

C-M-d Insert a byte with a code typed in decimal.

C-M-o Insert a byte with a code typed in octal.

C-M-x Insert a byte with a code typed in hex.

C-x [ Move to the beginning of a 1k-byte “page”.

C-x ] Move to the end of a 1k-byte “page”.

M-g Move to an address specified in hex.

M-j Move to an address specified in decimal.

C-c C-c Leave Hexl mode, going back to the major mode this buffer had before you invoked hexl-mode.

Other Hexl commands let you insert strings (sequences) of binary bytes, move by shorts or ints, etc.; type C-h a hexl-RET for details.

## 31.8 Saving Emacs Sessions

Use the desktop library to save the state of Emacs from one session to another. Once you save the Emacs *desktop*—the buffers, their file names, major modes, buffer positions, and so on—then subsequent Emacs sessions reload the saved desktop.

You can save the desktop manually with the command `M-x desktop-save`. You can also enable automatic saving of the desktop when you exit Emacs, and automatic restoration of the last saved desktop when Emacs starts: use the Customization buffer (see Section 33.1 [Easy Customization], page 686) to set `desktop-save-mode` to `t` for future sessions, or add this line in your init file (see Section 33.4 [Init File], page 711):

```
(desktop-save-mode 1)
```

If you turn on `desktop-save-mode` in your init file, then when Emacs starts, it looks for a saved desktop in the current directory. (More precisely, it looks in the directories specified by *desktop-path*, and uses the first desktop it finds.) Thus, you can have separate saved desktops in different directories, and the starting directory determines which one Emacs reloads. You can save the current desktop and reload one saved in another directory by typing `M-x desktop-change-dir`. Typing `M-x desktop-revert` reverts to the desktop previously reloaded.

Specify the option '`--no-desktop`' on the command line when you don't want it to reload any saved desktop. This turns off `desktop-save-mode` for the current session. Starting Emacs with the '`--no-init-file`' option also disables desktop reloading, since it bypasses the init file, where `desktop-save-mode` is usually turned on.

By default, all the buffers in the desktop are restored at one go. However, this may be slow if there are a lot of buffers in the desktop. You can specify the maximum number of buffers to restore immediately with the variable `desktop-restore-eager`; the remaining buffers are restored "lazily", when Emacs is idle.

Type `M-x desktop-clear` to empty the Emacs desktop. This kills all buffers except for internal ones, and clears the global variables listed in `desktop-globals-to-clear`. If you want this to preserve certain buffers, customize the variable `desktop-clear-preserve-buffers-regexp`, whose value is a regular expression matching the names of buffers not to kill.

If you want to save minibuffer history from one session to another, use the `savehist` library.

## 31.9 Recursive Editing Levels

A *recursive edit* is a situation in which you are using Emacs commands to perform arbitrary editing while in the middle of another Emacs command. For example, when you type `C-r` inside of a `query-replace`, you enter a recursive edit in which you can change the current buffer. On exiting from the recursive edit, you go back to the `query-replace`.

*Exiting* the recursive edit means returning to the unfinished command, which continues execution. The command to exit is `C-M-c` (`exit-recursive-edit`).

You can also *abort* the recursive edit. This is like exiting, but also quits the unfinished command immediately. Use the command `C-]` (`abort-recursive-edit`) to do this. See Section 34.1 [Quitting], page 717.

The mode line shows you when you are in a recursive edit by displaying square brackets around the parentheses that always surround the major and minor mode names. Every window's mode line shows this in the same way, since being in a recursive edit is true of Emacs as a whole rather than any particular window or buffer.

It is possible to be in recursive edits within recursive edits. For example, after typing `C-r` in a `query-replace`, you may type a command that enters the debugger. This begins a recursive editing level for the debugger, within the recursive editing level for `C-r`. Mode lines display a pair of square brackets for each recursive editing level currently in progress.

Exiting the inner recursive edit (such as with the debugger `c` command) resumes the command running in the next level up. When that command finishes, you can then use `C-M-c` to exit another recursive editing level, and so on. Exiting applies to the innermost level only. Aborting also gets out of only one level of recursive edit; it returns immediately to the command level of the previous recursive edit. If you wish, you can then abort the next recursive editing level.

Alternatively, the command `M-x top-level` aborts all levels of recursive edits, returning immediately to the top-level command reader. It also exits the minibuffer, if it is active.

The text being edited inside the recursive edit need not be the same text that you were editing at top level. It depends on what the recursive edit is for. If the command that invokes the recursive edit selects a different buffer first, that is the buffer you will edit recursively. In any case, you can switch buffers within the recursive edit in the normal manner (as long as the buffer-switching keys have not been rebound). You could probably do all the rest of your editing inside the recursive edit, visiting files and all. But this could have surprising effects (such as stack overflow) from time to time. So remember to exit or abort the recursive edit when you no longer need it.

In general, we try to minimize the use of recursive editing levels in GNU Emacs. This is because they constrain you to "go back" in a particular order—from the innermost level toward the top level. When possible, we present different activities in separate buffers so that you can switch between them as you please. Some commands switch to a new major mode which provides a command to switch back. These approaches give you more flexibility to go back to unfinished tasks in the order you choose.

## 31.10 Emulation

GNU Emacs can be programmed to emulate (more or less) most other editors. Standard facilities can emulate these:

CRiSP/Brief (PC editor)
: `M-x crisp-mode` enables key bindings to emulate the CRiSP/Brief editor. Note that this rebinds `M-x` to exit Emacs unless you set the variable `crisp-override-meta-x`. You can also use the command `M-x scroll-all-mode` or set the variable `crisp-load-scroll-all` to emulate CRiSP's scroll-all feature (scrolling all windows together).

EDT (DEC VMS editor)
: Turn on EDT emulation with `M-x edt-emulation-on`; restore normal command bindings with `M-x edt-emulation-off`.

  Most of the EDT emulation commands are keypad keys, and most standard Emacs key bindings are still available. The EDT emulation rebindings are done

in the global keymap, so there is no problem switching buffers or major modes while in EDT emulation.

TPU (DEC VMS editor)
: `M-x tpu-edt-on` turns on emulation of the TPU editor emulating EDT.

vi (Berkeley editor)
: Viper is the newest emulator for vi. It implements several levels of emulation; level 1 is closest to vi itself, while level 5 departs somewhat from strict emulation to take advantage of the capabilities of Emacs. To invoke Viper, type `M-x viper-mode`; it will guide you the rest of the way and ask for the emulation level. See Info file '`viper`', node '`Top`'.

vi (another emulator)
: `M-x vi-mode` enters a major mode that replaces the previously established major mode. All of the vi commands that, in real vi, enter "input" mode are programmed instead to return to the previous major mode. Thus, ordinary Emacs serves as vi's "input" mode.

  Because vi emulation works through major modes, it does not work to switch buffers during emulation. Return to normal Emacs first.

  If you plan to use vi emulation much, you probably want to bind a key to the `vi-mode` command.

vi (alternate emulator)
: `M-x vip-mode` invokes another vi emulator, said to resemble real vi more thoroughly than `M-x vi-mode`. "Input" mode in this emulator is changed from ordinary Emacs so you can use ESC to go back to emulated vi command mode. To get from emulated vi command mode back to ordinary Emacs, type `C-z`.

  This emulation does not work through major modes, and it is possible to switch buffers in various ways within the emulator. It is not so necessary to assign a key to the command `vip-mode` as it is with `vi-mode` because terminating insert mode does not use it.

  See Info file '`vip`', node '`Top`', for full information.

WordStar (old wordprocessor)
: `M-x wordstar-mode` provides a major mode with WordStar-like key bindings.

## 31.11 Hyperlinking and Navigation Features

The following subsections describe convenience features for handling URLs and other types of links occurring in Emacs buffer text.

### 31.11.1 Following URLs

`M-x browse-url RET` *`url`* `RET`
: Load a URL into a Web browser.

The Browse-URL package allows you to easily follow URLs from within Emacs. Most URLs are followed by invoking a web browser; '`mailto:`' URLs are followed by invoking the `compose-mail` Emacs command to send mail to the specified address (see Chapter 29 [Sending Mail], page 624).

The command `M-x browse-url` prompts for a URL, and follows it. If point is located near a plausible URL, that URL is offered as the default. The Browse-URL package also provides other commands which you might like to bind to keys, such as `browse-url-at-point` and `browse-url-at-mouse`.

You can customize Browse-URL's behavior via various options in the `browse-url` Customize group. In particular, the option `browse-url-mailto-function` lets you define how to follow '`mailto:`' URLs, while `browse-url-browser-function` lets you define how to follow other types of URLs. For more information, view the package commentary by typing `C-h P browse-url RET`.

### 31.11.2 Activating URLs

`M-x goto-address-mode`
: Activate URLs and e-mail addresses in the current buffer.

You can make Emacs mark out URLs specially in the current buffer, by typing `M-x goto-address-mode`. When this buffer-local minor mode is enabled, it finds all the URLs in the buffer, highlights them, and turns them into clickable buttons. You can follow the URL by typing `C-c RET` (`goto-address-at-point`) while point is on its text; or by clicking with `Mouse-2`, or by clicking `Mouse-1` quickly (see ⟨undefined⟩ [Mouse References], page ⟨undefined⟩). Following a URL is done by calling `browse-url` as a subroutine (see Section 31.11.1 [Browse-URL], page 677).

It can be useful to add `goto-address-mode` to mode hooks and hooks for displaying an incoming message (e.g., `rmail-show-message-hook` for Rmail, and `mh-show-mode-hook` for MH-E). This is not needed for Gnus, which has a similar feature of its own.

### 31.11.3 Finding Files and URLs at Point

The FFAP package replaces certain key bindings for finding files, such as `C-x C-f`, with commands that provide more sensitive defaults. These commands behave like the ordinary ones when given a prefix argument. Otherwise, they get the default file name or URL from the text around point. If what is found in the buffer has the form of a URL rather than a file name, the commands use `browse-url` to view it (see Section 31.11.1 [Browse-URL], page 677).

This feature is useful for following references in mail or news buffers, '`README`' files, '`MANIFEST`' files, and so on. For more information, view the package commentary by typing `C-h P ffap RET`.

To enable FFAP, type `M-x ffap-bindings`. This makes the following key bindings, and also installs hooks for additional FFAP functionality in Rmail, Gnus and VM article buffers.

`C-x C-f` *filename* `RET`
: Find *filename*, guessing a default from text around point (`find-file-at-point`).

`C-x C-r`
: `ffap-read-only`, analogous to `find-file-read-only`.

`C-x C-v`
: `ffap-alternate-file`, analogous to `find-alternate-file`.

`C-x d` *directory* `RET`
: Start Dired on *directory*, defaulting to the directory name at point (`dired-at-point`).

`C-x C-d` `ffap-list-directory`, analogous to `list-directory`.

`C-x 4 f` `ffap-other-window`, analogous to `find-file-other-window`.

`C-x 4 r` `ffap-read-only-other-window`, analogous to `find-file-read-only-other-window`.

`C-x 4 d` `ffap-dired-other-window`, like `dired-other-window`.

`C-x 5 f` `ffap-other-frame`, analogous to `find-file-other-frame`.

`C-x 5 r` `ffap-read-only-other-frame`, analogous to `find-file-read-only-other-frame`.

`C-x 5 d` `ffap-dired-other-frame`, analogous to `dired-other-frame`.

`M-x ffap-next`
Search buffer for next file name or URL, then find that file or URL.

`S-Mouse-3`
`ffap-at-mouse` finds the file guessed from text around the position of a mouse click.

`C-S-Mouse-3`
Display a menu of files and URLs mentioned in current buffer, then find the one you select (`ffap-menu`).

## 31.12 Other Amusements

The `animate` package makes text dance (e.g., `M-x animate-birthday-present`).

`M-x blackbox`, `M-x mpuz` and `M-x 5x5` are puzzles. `blackbox` challenges you to determine the location of objects inside a box by tomography. `mpuz` displays a multiplication puzzle with letters standing for digits in a code that you must guess—to guess a value, type a letter and then the digit you think it stands for. The aim of `5x5` is to fill in all the squares.

`M-x bubbles` is a game in which the object is to remove as many bubbles as you can in the smallest number of moves.

`M-x decipher` helps you to cryptanalyze a buffer which is encrypted in a simple monoalphabetic substitution cipher.

`M-x dissociated-press` scrambles the text in the current Emacs buffer, word by word or character by character, writing its output to a buffer named '`*Dissociation*`'. A positive argument tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word, and specifies the number of overlap words. Dissociated Press produces results fairly like those of a Markov chain, but is however, an independent, ignoriginal invention; it techniquitously copies several consecutive characters from the sample text between random jumps, unlike a Markov chain which would jump randomly after each word or character. Keep dissociwords out of your documentation, if you want it to be well userenced and properbose.

`M-x dunnet` runs an text-based adventure game.

If you want a little more personal involvement, try `M-x gomoku`, which plays the game Go Moku with you.

If you are a little bit bored, you can try `M-x hanoi`. If you are considerably bored, give it a numeric argument. If you are very, very bored, try an argument of 9. Sit back and watch.

`M-x life` runs Conway's "Life" cellular automaton.

`M-x landmark` runs a relatively non-participatory game in which a robot attempts to maneuver towards a tree at the center of the window based on unique olfactory cues from each of the four directions.

`M-x morse-region` converts the text in the region to Morse code; `M-x unmorse-region` converts it back. `M-x nato-region` converts the text in the region to NATO phonetic alphabet; `M-x denato-region` converts it back.

`M-x pong`, `M-x snake` and `M-x tetris` are implementations of the well-known Pong, Snake and Tetris games.

`M-x solitaire` plays a game of solitaire in which you jump pegs across other pegs.

The command `M-x zone` plays games with the display when Emacs is idle.

Finally, if you find yourself frustrated, try describing your problems to the famous psychotherapist Eliza. Just do `M-x doctor`. End each input by typing RET twice.

# 32 Preparing Lisp code for distribution

Emacs provides a standard way to distribute Emacs Lisp code to users. A *package* is a collection of one or more files, formatted and bundled in such a way that users can easily download, install, uninstall, and upgrade it.

The following sections describe how to create a package, and how to put it in a *package archive* for others to download. See Section "Packages" in *The GNU Emacs Manual*, for a description of user-level features of the packaging system.

## 32.1 Packaging Basics

A package is either a *simple package* or a *multi-file package*. A simple package is stored in a package archive as a single Emacs Lisp file, while a multi-file package is stored as a tar file (containing multiple Lisp files, and possibly non-Lisp files such as a manual).

In ordinary usage, the difference between simple packages and multi-file packages is relatively unimportant; the Package Menu interface makes no distinction between them. However, the procedure for creating them differs, as explained in the following sections.

Each package (whether simple or multi-file) has certain *attributes*:

Name
: A short word (e.g., '`auctex`'). This is usually also the symbol prefix used in the program (see ⟨undefined⟩ [Coding Conventions], page ⟨undefined⟩).

Version
: A version number, in a form that the function `version-to-list` understands (e.g., '`11.86`'). Each release of a package should be accompanied by an increase in the version number.

Brief description
: This is shown when the package is listed in the Package Menu. It should occupy a single line, ideally in 36 characters or less.

Long description
: This is shown in the buffer created by `C-h P` (`describe-package`), following the package's brief description and installation status. It normally spans multiple lines, and should fully describe the package's capabilities and how to begin using it once it is installed.

Dependencies
: A list of other packages (possibly including minimal acceptable version numbers) on which this package depends. The list may be empty, meaning this package has no dependencies. Otherwise, installing this package also automatically installs its dependencies; if any dependency cannot be found, the package cannot be installed.

Installing a package, either via the command `package-install-file`, or via the Package Menu, creates a subdirectory of `package-user-dir` named '*`name-version`*', where *name* is the package's name and *version* its version (e.g., '`~/.emacs.d/elpa/auctex-11.86/`'). We call this the package's *content directory*. It is where Emacs puts the package's contents (the single Lisp file for a simple package, or the files extracted from a multi-file package).

Emacs then searches every Lisp file in the content directory for autoload magic comments (see ⟨undefined⟩ [Autoload], page ⟨undefined⟩). These autoload definitions are saved to a file

named '*name*`-autoloads.el`' in the content directory. They are typically used to autoload the principal user commands defined in the package, but they can also perform other tasks, such as adding an element to `auto-mode-alist` (see Section 20.2.2 [Auto Major Mode], page 403). Note that a package typically does *not* autoload every function and variable defined within it—only the handful of commands typically called to begin using the package. Emacs then byte-compiles every Lisp file in the package.

After installation, the installed package is *loaded*: Emacs adds the package's content directory to `load-path`, and evaluates the autoload definitions in '*name*`-autoloads.el`'.

Whenever Emacs starts up, it automatically calls the function `package-initialize` to load installed packages. This is done after loading the init file and abbrev file (if any) and before running `after-init-hook` (see ⟨undefined⟩ [Startup Summary], page ⟨undefined⟩). Automatic package loading is disabled if the user option `package-enable-at-startup` is `nil`.

`package-initialize` **&optional** *no-activate* [Command]

This function initializes Emacs' internal record of which packages are installed, and loads them. The user option `package-load-list` specifies which packages to load; by default, all installed packages are loaded. See Section "Package Installation" in *The GNU Emacs Manual*.

The optional argument *no-activate*, if non-`nil`, causes Emacs to update its record of installed packages without actually loading them; it is for internal use only.

## 32.2 Simple Packages

A simple package consists of a single Emacs Lisp source file. The file must conform to the Emacs Lisp library header conventions (see ⟨undefined⟩ [Library Headers], page ⟨undefined⟩). The package's attributes are taken from the various headers, as illustrated by the following example:

```
;;; superfrobnicator.el --- Frobnicate and bifurcate flanges

;; Copyright (C) 2011 Free Software Foundation, Inc.

;; Author: J. R. Hacker <jrh@example.com>
;; Version: 1.3
;; Package-Requires: ((flange "1.0"))
;; Keywords: frobnicate

...

;;; Commentary:

;; This package provides a minor mode to frobnicate and/or
;; bifurcate any flanges you desire.  To activate it, just type
...

;;;###autoload
(define-minor-mode superfrobnicator-mode
```

```
...
```

The name of the package is the same as the base name of the file, as written on the first line. Here, it is '`superfrobnicator`'.

The brief description is also taken from the first line. Here, it is '`Frobnicate and bifurcate flanges`'.

The version number comes from the '`Package-Version`' header, if it exists, or from the '`Version`' header otherwise. One or the other *must* be present. Here, the version number is 1.3.

If the file has a '`;;; Commentary:`' section, this section is used as the long description. (When displaying the description, Emacs omits the '`;;; Commentary:`' line, as well as the leading comment characters in the commentary itself.)

If the file has a '`Package-Requires`' header, that is used as the package dependencies. In the above example, the package depends on the '`flange`' package, version 1.0 or higher. See ⟨undefined⟩ [Library Headers], page ⟨undefined⟩, for a description of the '`Package-Requires`' header. If the header is omitted, the package has no dependencies.

The file ought to also contain one or more autoload magic comments, as explained in Section 32.1 [Packaging Basics], page 681. In the above example, a magic comment autoloads `superfrobnicator-mode`.

See Section 32.4 [Package Archives], page 684, for a explanation of how to add a single-file package to a package archive.

## 32.3 Multi-file Packages

A multi-file package is less convenient to create than a single-file package, but it offers more features: it can include multiple Emacs Lisp files, an Info manual, and other file types (such as images).

Prior to installation, a multi-file package is stored in a package archive as a tar file. The tar file must be named '*`name`*`-`*`version`*`.tar`', where *name* is the package name and *version* is the version number. Its contents, once extracted, must all appear in a directory named '*`name`*`-`*`version`*', the *content directory* (see Section 32.1 [Packaging Basics], page 681). Files may also extract into subdirectories of the content directory.

One of the files in the content directory must be named '*`name`*`-pkg.el`'. It must contain a single Lisp form, consisting of a call to the function `define-package`, described below. This defines the package's version, brief description, and requirements.

For example, if we distribute version 1.3 of the superfrobnicator as a multi-file package, the tar file would be '`superfrobnicator-1.3.tar`'. Its contents would extract into the directory '`superfrobnicator-1.3`', and one of these would be the file '`superfrobnicator-pkg.el`'.

`define-package` *name version* **&optional** *docstring requirements* [Function]

This function defines a package. *name* is the package name, a string. *version* is the version, as a string of a form that can be understood by the function `version-to-list`. *docstring* is the brief description.

*requirements* is a list of required packages and their versions. Each element in this list should have the form `(`*`dep-name dep-version`*`)`, where *dep-name* is a symbol whose

name is the dependency's package name, and *dep-version* is the dependency's version (a string).

If the content directory contains a file named '`README`', this file is used as the long description.

If the content directory contains a file named '`dir`', this is assumed to be an Info directory file made with `install-info`. See Section "Invoking install-info" in *Texinfo*. The relevant Info files should also be present in the content directory. In this case, Emacs will automatically add the content directory to `Info-directory-list` when the package is activated.

Do not include any '`.elc`' files in the package. Those are created when the package is installed. Note that there is no way to control the order in which files are byte-compiled.

Do not include any file named '*name*`-autoloads.el`'. This file is reserved for the package's autoload definitions (see Section 32.1 [Packaging Basics], page 681). It is created automatically when the package is installed, by searching all the Lisp files in the package for autoload magic comments.

If the multi-file package contains auxiliary data files (such as images), the package's Lisp code can refer to these files via the variable `load-file-name` (see ⟨undefined⟩ [Loading], page ⟨undefined⟩). Here is an example:

```
(defconst superfrobnicator-base (file-name-directory load-file-name))

(defun superfrobnicator-fetch-image (file)
  (expand-file-name file superfrobnicator-base))
```

## 32.4 Creating and Maintaining Package Archives

Via the Package Menu, users may download packages from *package archives*. Such archives are specified by the variable `package-archives`, whose default value contains a single entry: the archive hosted by the GNU project at `elpa.gnu.org`. This section describes how to set up and maintain a package archive.

`package-archives` [User Option]

The value of this variable is an alist of package archives recognized by the Emacs package manager.

Each alist element corresponds to one archive, and should have the form `(`*`id . location`*`)`, where *id* is the name of the archive (a string) and *location* is its *base location* (a string).

If the base location starts with '`http:`', it is treated as a HTTP URL, and packages are downloaded from this archive via HTTP (as is the case for the default GNU archive).

Otherwise, the base location should be a directory name. In this case, Emacs retrieves packages from this archive via ordinary file access. Such "local" archives are mainly useful for testing.

A package archive is simply a directory in which the package files, and associated files, are stored. If you want the archive to be reachable via HTTP, this directory must be accessible to a web server. How to accomplish this is beyond the scope of this manual.

A convenient way to set up and update a package archive is via the `package-x` library. This is included with Emacs, but not loaded by default; type `M-x load-library RET package-x RET` to load it, or add `(require 'package-x)` to your init file. See Section "Lisp Libraries" in *The GNU Emacs Manual*. Once loaded, you can make use of the following:

`package-archive-upload-base` [User Option]

The value of this variable is the base location of a package archive, as a directory name. The commands in the `package-x` library will use this base location.

The directory name should be absolute. You may specify a remote name, such as '`/ssh:foo@example.com:/var/www/packages/`', if the package archive is on a different machine. See Section "Remote Files" in *The GNU Emacs Manual*.

`package-upload-file` *filename* [Command]

This command prompts for *filename*, a file name, and uploads that file to `package-archive-upload-base`. The file must be either a simple package (a '`.el`' file) or a multi-file package (a '`.tar`' file); otherwise, an error is raised. The package attributes are automatically extracted, and the archive's contents list is updated with this information.

If `package-archive-upload-base` does not specify a valid directory, the function prompts interactively for one. If the directory does not exist, it is created. The directory need not have any initial contents (i.e., you can use this command to populate an initially empty archive).

`package-upload-buffer` [Command]

This command is similar to `package-upload-file`, but instead of prompting for a package file, it uploads the contents of the current buffer. The current buffer must be visiting a simple package (a '`.el`' file) or a multi-file package (a '`.tar`' file); otherwise, an error is raised.

After you create an archive, remember that it is not accessible in the Package Menu interface unless it is in `package-archives`.

# 33 Customization

This chapter describes some simple methods to customize the behavior of Emacs.

Apart from the methods described here, see Appendix F [X Resources], page 769 for information about using X resources to customize Emacs, and see Chapter 14 [Keyboard Macros], page 222 for information about recording and replaying keyboard macros. Making more far-reaching and open-ended changes involves writing Emacs Lisp code; see *The Emacs Lisp Reference Manual*.

## 33.1 Easy Customization Interface

Emacs has many *settings* which you can change. Most settings are *customizable variables* (see Section 33.2 [Variables], page 694), which are also called *user options*. There is a huge number of customizable variables, controlling numerous aspects of Emacs behavior; the variables documented in this manual are listed in [Variable Index], page 845. A separate class of settings are the *faces*, which determine the fonts, colors, and other attributes of text (see Section 11.12 [Faces], page 137).

To browse and alter settings (both variables and faces), type `M-x customize`. This creates a *customization buffer*, which lets you navigate through a logically organized list of settings, edit and set their values, and save them permanently.

### 33.1.1 Customization Groups

Customization settings are organized into *customization groups*. These groups are collected into bigger groups, all the way up to a master group called `Emacs`.

`M-x customize` creates a customization buffer that shows the top-level `Emacs` group. It looks like this, in part:

```
To apply changes, use the Save or Set buttons.
For details, see [Saving Customizations] in the [Emacs manual].

---------------------------------------- [ Search ]

 Operate on all settings in this buffer:
 [ Set for current session ] [ Save for future sessions ]
 [ Undo edits ] [ Reset to saved ] [ Erase customizations ] [ Exit ]


Emacs group: Customization of the One True Editor.
      [State]: visible group members are all at standard values.
      See also [Manual].

[Editing] : Basic text editing facilities.

[Convenience] : Convenience features for faster editing.

more second-level groups
```

The main part of this buffer shows the '`Emacs`' customization group, which contains several other groups ('`Editing`', '`Convenience`', etc.). The contents of those groups are not listed here, only one line of documentation each.

The *state* of the group indicates whether setting in that group has been edited, set or saved. See Section 33.1.3 [Changing a Variable], page 687.

Most of the customization buffer is read-only, but it includes some *editable fields* that you can edit. For example, at the top of the customization buffer is an editable field for searching for settings (see Section 33.1.2 [Browsing Custom], page 687). There are also *buttons* and *links*, which you can activate by either clicking with the mouse, or moving point there and typing RET. For example, the group names like '`[Editing]`' are links; activating one of these links brings up the customization buffer for that group.

In the customizable buffer, you can type TAB (`widget-forward`) to move forward to the next button or editable field. `S-TAB` (`widget-backward`) moves back to the previous button or editable field.

### 33.1.2 Browsing and Searching for Settings

From the top-level customization buffer created by `M-x customize`, you can follow the links to the subgroups of the '`Emacs`' customization group. These subgroups may contain settings for you to customize; they may also contain further subgroups, dealing with yet more specialized subsystems of Emacs. As you navigate the hierarchy of customization groups, you should find some settings that you want to customize.

If you are interested in customizing a particular setting or customization group, you can go straight there with the commands `M-x customize-option`, `M-x customize-face`, or `M-x customize-group`. See Section 33.1.6 [Specific Customization], page 691.

If you don't know exactly what groups or settings you want to customize, you can search for them using the editable search field at the top of each customization buffer. Here, you can type in a search term—either one or more words separated by spaces, or a regular expression (see Section 12.6 [Regexps], page 206). Then type RET in the field, or activate the '`Search`' button next to it, to switch to a customization buffer containing groups and settings that match those terms. Note, however, that this feature only finds groups and settings that are loaded in the current Emacs session.

If you don't want customization buffers to show the search field, change the variable `custom-search-field` to `nil`.

The command `M-x customize-apropos` is similar to using the search field, except that it reads the search term(s) using the minibuffer. See Section 33.1.6 [Specific Customization], page 691.

`M-x customize-browse` is another way to browse the available settings. This command creates a special customization buffer which shows only the names of groups and settings, in a structured layout. You can show the contents of a group, in the same buffer, by invoking the '`[+]`' button next to the group name. When the group contents are shown, the button changes to '`[-]`'; invoking that hides the group contents again. Each group or setting in this buffer has a link which says '`[Group]`', '`[Option]`' or '`[Face]`'. Invoking this link creates an ordinary customization buffer showing just that group, option, or face; this is the way to change settings that you find with `M-x customize-browse`.

### 33.1.3 Changing a Variable

Here is an example of what a variable, or user option, looks like in the customization buffer:

```
[Hide] Kill Ring Max: 60
   [State]: STANDARD.
   Maximum length of kill ring before oldest elements are thrown away.
```

The first line shows that the variable is named `kill-ring-max`, formatted as '`Kill Ring Max`' for easier viewing. Its value is '`60`'. The button labeled '`[Hide]`', if activated, hides the variable's value and state; this is useful to avoid cluttering up the customization buffer with very long values (for this reason, variables that have very long values may start out hidden). If you use the '`[Hide]`' button, it changes to '`[Show Value]`', which you can activate to reveal the value and state. On a graphical display, the '`[Hide]`' and '`[Show Value]`' buttons are replaced with graphical triangles pointing downwards and rightwards respectively.

The line after the variable name indicates the *customization state* of the variable: in this example, '`STANDARD`' means you have not changed the variable, so its value is the default one. The '`[State]`' button gives a menu of operations for customizing the variable.

Below the customization state is the documentation for the variable. This is the same documentation that would be shown by the `C-h v` command (see Section 33.2.1 [Examining], page 695). If the documentation is more than one line long, only one line may be shown. If so, that line ends with a '`[More]`' button; activate this to see the full documentation.

To enter a new value for '`Kill Ring Max`', just move point to the value and edit it. For example, type `M-d` to delete the '`60`' and type in another number. As you begin to alter the text, the '`[State]`' line will change:

```
[State]: EDITED, shown value does not take effect until you
         set or save it.
```

Editing the value does not make it take effect right away. To do that, you must *set* the variable by activating the '`[State]`' button and choosing '`Set for Current Session`'. Then the variable's state becomes:

```
[State]: SET for current session only.
```

You don't have to worry about specifying a value that is not valid; the '`Set for Current Session`' operation checks for validity and will not install an unacceptable value.

While editing certain kinds of values, such as file names, directory names, and Emacs command names, you can perform completion with `C-M-i` (`widget-complete`), or the equivalent keys `M-TAB` or `ESC TAB`. This behaves much like minibuffer completion (see Section 5.4 [Completion], page 70).

Typing RET on an editable value field moves point forward to the next field or button, like TAB. You can thus type RET when you are finished editing a field, to move on to the next button or field. To insert a newline within an editable field, use `C-o` or `C-q C-j`.

For some variables, there is only a fixed set of legitimate values, and you are not allowed to edit the value directly. Instead, a '`[Value Menu]`' button appears before the value; activating this button presents a choice of values. For a boolean "on or off" value, the button says '`[Toggle]`', and flips the value. After using the '`[Value Menu]`' or '`[Toggle]`' button, you must again set the variable to make the chosen value take effect.

Some variables have values with complex structure. For example, the value of `minibuffer-frame-alist` is an association list. Here is how it appears in the customization buffer:

```
[Hide] Minibuffer Frame Alist:
[INS] [DEL] Parameter: width
            Value: 80
[INS] [DEL] Parameter: height
            Value: 2
```

```
[INS]
   [ State ]: STANDARD.
   Alist of parameters for the initial minibuffer frame. [Hide]
   [. . . more lines of documentation. . . ]
```

In this case, each association in the list consists of two items, one labeled '`Parameter`' and one labeled '`Value`'; both are editable fields. You can delete an association from the list with the '`[DEL]`' button next to it. To add an association, use the '`[INS]`' button at the position where you want to insert it; the very last '`[INS]`' button inserts at the end of the list.

When you set a variable, the new value takes effect only in the current Emacs session. To *save* the value for future sessions, use the '`[State]`' button and select the '`Save for Future Sessions`' operation. See Section 33.1.4 [Saving Customizations], page 690.

You can also restore the variable to its standard value by using the '`[State]`' button and selecting the '`Erase Customization`' operation. There are actually four reset operations:

'`Undo Edits`'
: If you have modified but not yet set the variable, this restores the text in the customization buffer to match the actual value.

'`Reset to Saved`'
: This restores the value of the variable to the last saved value, and updates the text accordingly.

'`Erase Customization`'
: This sets the variable to its standard value. Any saved value that you have is also eliminated.

'`Set to Backup Value`'
: This sets the variable to a previous value that was set in the customization buffer in this session. If you customize a variable and then reset it, which discards the customized value, you can get the discarded value back again with this operation.

Sometimes it is useful to record a comment about a specific customization. Use the '`Add Comment`' item from the '`[State]`' menu to create a field for entering the comment.

Near the top of the customization buffer are two lines of buttons:

```
[Set for Current Session] [Save for Future Sessions]
[Undo Edits] [Reset to Saved] [Erase Customization]   [Exit]
```

Each of the first five buttons performs the stated operation—set, save, reset, etc.—on all the settings in the buffer that could meaningfully be affected. They do not operate on settings that are hidden, nor on subgroups that are hidden or not visible in the buffer.

The command `C-c C-c` (`Custom-set`) is equivalent using to the '`[Set for Current Session]`' button. The command `C-x C-s` (`Custom-save`) is like using the '`[Save for Future Sessions]`' button.

The '`[Exit]`' button switches out of the customization buffer, and buries the buffer at the bottom of the buffer list. To make it kill the customization buffer instead, change the variable `custom-buffer-done-kill` to `t`.

### 33.1.4 Saving Customizations

In the customization buffer, you can *save* a customization setting by choosing the '`Save for Future Sessions`' choice from its '`[State]`' button. The `C-x C-s` (`Custom-save`) command, or the '`[Save for Future Sessions]`' button at the top of the customization buffer, saves all applicable settings in the buffer.

Saving works by writing code to a file, usually your initialization file (see Section 33.4 [Init File], page 711). Future Emacs sessions automatically read this file at startup, which sets up the customizations again.

You can choose to save customizations somewhere other than your initialization file. To make this work, you must add a couple of lines of code to your initialization file, to set the variable `custom-file` to the name of the desired file, and to load that file. For example:

```
(setq custom-file "~/.emacs-custom.el")
(load custom-file)
```

You can even specify different customization files for different Emacs versions, like this:

```
(cond ((< emacs-major-version 22)
       ;; Emacs 21 customization.
       (setq custom-file "~/.custom-21.el"))
      ((and (= emacs-major-version 22)
            (< emacs-minor-version 3))
       ;; Emacs 22 customization, before version 22.3.
       (setq custom-file "~/.custom-22.el"))
      (t
       ;; Emacs version 22.3 or later.
       (setq custom-file "~/.emacs-custom.el")))

(load custom-file)
```

If Emacs was invoked with the '`-q`' or '`--no-init-file`' options (see Section E.2 [Initial Options], page 755), it will not let you save your customizations in your initialization file. This is because saving customizations from such a session would wipe out all the other customizations you might have on your initialization file.

### 33.1.5 Customizing Faces

You can customize faces (see Section 11.12 [Faces], page 137), which determine how Emacs displays different types of text. Customization groups can contain both variables and faces.

For example, in programming language modes, source code comments are shown with `font-lock-comment-face` (see ⟨undefined⟩ [Font Lock], page ⟨undefined⟩). In a customization buffer, that face appears like this:

```
[Hide] Font Lock Comment Face:[sample]
   [State] : STANDARD.
   Font Lock mode face used to highlight comments.
   [ ] Font Family: --
   [ ] Font Foundry: --
   [ ] Width: --
   [ ] Height: --
   [ ] Weight: --
   [ ] Slant: --
   [ ] Underline: --
```

```
[ ] Overline: --
[ ] Strike-through: --
[ ] Box around text: --
[ ] Inverse-video: --
[X] Foreground: Firebrick     [Choose]  (sample)
[ ] Background: --
[ ] Stipple: --
[ ] Inherit: --
[Hide Unused Attributes]
```

The first three lines show the name, '`[State]`' button, and documentation for the face. Below that is a list of *face attributes*. In front of each attribute is a checkbox. A filled checkbox, '`[X]`', means that the face specifies a value for this attribute; an empty checkbox, '`[ ]`', means that the face does not specify any special value for the attribute. You can activate a checkbox to specify or unspecify its attribute.

A face does not have to specify every single attribute; in fact, most faces only specify a few attributes. In the above example, `font-lock-comment-face` only specifies the foreground color. Any unspecified attribute is taken from the special face named `default`, whose attributes are all specified. The `default` face is the face used to display any text that does not have an explicitly-assigned face; furthermore, its background color attribute serves as the background color of the frame.

The '`Hide Unused Attributes`' button, at the end of the attribute list, hides the unspecified attributes of the face. When attributes are being hidden, the button changes to '`[Show All Attributes]`', which reveals the entire attribute list. The customization buffer may start out with unspecified attributes hidden, to avoid cluttering the interface.

When an attribute is specified, you can change its value in the usual ways.

Foreground and background colors can be specified using either color names or RGB triplets (see ⟨undefined⟩ [Colors], page ⟨undefined⟩). You can also use the '`[Choose]`' button to switch to a list of color names; select a color with RET in that buffer to put the color name in the value field.

Setting, saving and resetting a face work like the same operations for variables (see Section 33.1.3 [Changing a Variable], page 687).

A face can specify different appearances for different types of displays. For example, a face can make text red on a color display, but use a bold font on a monochrome display. To specify multiple appearances for a face, select '`For All Kinds of Displays`' in the menu you get from invoking '`[State]`'.

### 33.1.6 Customizing Specific Items

`M-x customize-option` RET *option* RET
`M-x customize-variable` RET *option* RET
: Set up a customization buffer for just one user option, *option*.

`M-x customize-face` RET *face* RET
: Set up a customization buffer for just one face, *face*.

`M-x customize-group` RET *group* RET
: Set up a customization buffer for just one group, *group*.

`M-x customize-apropos` RET *regexp* RET
: Set up a customization buffer for all the settings and groups that match *regexp*.

`M-x customize-changed RET` *version* `RET`
: Set up a customization buffer with all the settings and groups whose meaning has changed since Emacs version *version*.

`M-x customize-saved`
: Set up a customization buffer containing all settings that you have saved with customization buffers.

`M-x customize-unsaved`
: Set up a customization buffer containing all settings that you have set but not saved.

If you want to customize a particular user option, type `M-x customize-option`. This reads the variable name, and sets up the customization buffer with just that one user option. When entering the variable name into the minibuffer, completion is available, but only for the names of variables that have been loaded into Emacs.

Likewise, you can customize a specific face using `M-x customize-face`. You can set up a customization buffer for a specific customization group using `M-x customize-group`.

`M-x customize-apropos` prompts for a search term—either one or more words separated by spaces, or a regular expression—and sets up a customization buffer for all *loaded* settings and groups with matching names. This is like using the search field at the top of the customization buffer (see Section 33.1.1 [Customization Groups], page 686).

When you upgrade to a new Emacs version, you might want to consider customizing new settings, and settings whose meanings or default values have changed. To do this, use `M-x customize-changed` and specify a previous Emacs version number using the minibuffer. It creates a customization buffer which shows all the settings and groups whose definitions have been changed since the specified version, loading them if necessary.

If you change settings and then decide the change was a mistake, you can use two commands to revisit your changes. Use `M-x customize-saved` to customize settings that you have saved. Use `M-x customize-unsaved` to customize settings that you have set but not saved.

### 33.1.7 Custom Themes

*Custom themes* are collections of settings that can be enabled or disabled as a unit. You can use Custom themes to switch easily between various collections of settings, and to transfer such collections from one computer to another.

A Custom theme is stored as an Emacs Lisp source file. If the name of the Custom theme is *name*, the theme file is named '*name*`-theme.el`'. See Section 33.1.8 [Creating Custom Themes], page 693, for the format of a theme file and how to make one.

Type `M-x customize-themes` to switch to a buffer named '`*Custom Themes*`', which lists the Custom themes that Emacs knows about. By default, Emacs looks for theme files in two locations: the directory specified by the variable `custom-theme-directory` (which defaults to '`~/.emacs.d/`'), and a directory named '`etc/themes`' in your Emacs installation (see the variable `data-directory`). The latter contains several Custom themes which are distributed with Emacs, which customize Emacs's faces to fit various color schemes. (Note, however, that Custom themes need not be restricted to this purpose; they can be used to customize variables too).

If you want Emacs to look for Custom themes in some other directory, add the directory name to the list variable `custom-theme-load-path`. Its default value is `(custom-theme-directory t)`; here, the symbol `custom-theme-directory` has the special meaning of the value of the variable `custom-theme-directory`, while `t` stands for the built-in theme directory '`etc/themes`'. The themes listed in the '`*Custom Themes*`' buffer are those found in the directories specified by `custom-theme-load-path`.

In the '`*Custom Themes*`' buffer, you can activate the checkbox next to a Custom theme to enable or disable the theme for the current Emacs session. When a Custom theme is enabled, all of its settings (variables and faces) take effect in the Emacs session. To apply the choice of theme(s) to future Emacs sessions, type `C-x C-s` (`custom-theme-save`) or use the '`[Save Theme Settings]`' button.

When you first enable a Custom theme, Emacs displays the contents of the theme file and asks if you really want to load it. Because loading a Custom theme can execute arbitrary Lisp code, you should only say yes if you know that the theme is safe; in that case, Emacs offers to remember in the future that the theme is safe (this is done by saving the theme file's SHA-256 hash to the variable `custom-safe-themes`; if you want to treat all themes as safe, change its value to `t`). Themes that come with Emacs (in the '`etc/themes`' directory) are exempt from this check, and are always considered safe.

Setting or saving Custom themes actually works by customizing the variable `custom-enabled-themes`. The value of this variable is a list of Custom theme names (as Lisp symbols, e.g., `tango`). Instead of using the '`*Custom Themes*`' buffer to set `custom-enabled-themes`, you can customize the variable using the usual customization interface, e.g., with `M-x customize-option`. Note that Custom themes are not allowed to set `custom-enabled-themes` themselves.

Any customizations that you make through the customization buffer take precedence over theme settings. This lets you easily override individual theme settings that you disagree with. If settings from two different themes overlap, the theme occurring earlier in `custom-enabled-themes` takes precedence. In the customization buffer, if a setting has been changed from its default by a Custom theme, its '`State`' display shows '`THEMED`' instead of '`STANDARD`'.

You can enable a specific Custom theme in the current Emacs session by typing `M-x load-theme`. This prompts for a theme name, loads the theme from the theme file, and enables it. If a theme file has been loaded before, you can enable the theme without loading its file by typing `M-x enable-theme`. To disable a Custom theme, type `M-x disable-theme`.

To see a description of a Custom theme, type `?` on its line in the '`*Custom Themes*`' buffer; or type `M-x describe-theme` anywhere in Emacs and enter the theme name.

### 33.1.8 Creating Custom Themes

You can define a Custom theme using an interface similar to the customization buffer, by typing `M-x customize-create-theme`. This switches to a buffer named '`*Custom Theme*`'. It also offers to insert some common Emacs faces into the theme (a convenience, since Custom themes are often used to customize faces). If you answer no, the theme will initially contain no settings.

Near the top of the '`*Custom Theme*`' buffer are editable fields where you can enter the theme's name and description. The name can be anything except '`user`'. The description

is the one that will be shown when you invoke `M-x describe-theme` for the theme. Its first line should be a brief one-sentence summary; in the buffer made by `M-x customize-themes`, this sentence is displayed next to the theme name.

To add a new setting to the theme, use the '`[Insert Additional Face]`' or '`[Insert Additional Variable]`' buttons. Each button reads a face or variable name using the minibuffer, with completion, and inserts a customization entry for the face or variable. You can edit the variable values or face attributes in the same way as in a normal customization buffer. To remove a face or variable from the theme, uncheck the checkbox next to its name.

After specifying the Custom theme's faces and variables, type `C-x C-s` (`custom-theme-write`) or use the buffer's '`[Save Theme]`' button. This saves the theme file, named '*name*`-theme.el`' where *name* is the theme name, in the directory named by `custom-theme-directory`.

From the '`*Custom Theme*`' buffer, you can view and edit an existing Custom theme by activating the '`[Visit Theme]`' button and specifying the theme name. You can also add the settings of another theme into the buffer, using the '`[Merge Theme]`' button. You can import your non-theme settings into a Custom theme by using the '`[Merge Theme]`' button and specifying the special theme named '`user`'.

A theme file is simply an Emacs Lisp source file, and loading the Custom theme works by loading the Lisp file. Therefore, you can edit a theme file directly instead of using the '`*Custom Theme*`' buffer. See Section "Custom Themes" in *The Emacs Lisp Reference Manual*, for details.

## 33.2 Variables

A *variable* is a Lisp symbol which has a value. The symbol's name is also called the *variable name*. A variable name can contain any characters that can appear in a file, but most variable names consist of ordinary words separated by hyphens.

The name of the variable serves as a compact description of its role. Most variables also have a *documentation string*, which describes what the variable's purpose is, what kind of value it should have, and how the value will be used. You can view this documentation using the help command `C-h v` (`describe-variable`). See Section 33.2.1 [Examining], page 695.

Emacs uses many Lisp variables for internal record keeping, but the most interesting variables for a non-programmer user are those meant for users to change—these are called *customizable variables* or *user options* (see Section 33.1 [Easy Customization], page 686). In the following sections, we will describe other aspects of Emacs variables, such as how to set them outside Customize.

Emacs Lisp allows any variable (with a few exceptions) to have any kind of value. However, many variables are meaningful only if assigned values of a certain type. For example, only numbers are meaningful values for `kill-ring-max`, which specifies the maximum length of the kill ring (see Section 9.2.2 [Earlier Kills], page 98); if you give `kill-ring-max` a string value, commands such as `C-y` (`yank`) will signal an error. On the other hand, some variables don't care about type; for instance, if a variable has one effect for `nil` values and another effect for "non-`nil`" values, then any value that is not the symbol `nil` induces the second effect, regardless of its type (by convention, we usually use the value `t`—a symbol which stands for "true"—to specify a non-`nil` value). If you set a variable using the

customization buffer, you need not worry about giving it an invalid type: the customization buffer usually only allows you to enter meaningful values. When in doubt, use `C-h v` (`describe-variable`) to check the variable's documentation string to see kind of value it expects (see Section 33.2.1 [Examining], page 695).

### 33.2.1 Examining and Setting Variables

`C-h v` *var* `RET`
: Display the value and documentation of variable *var* (`describe-variable`).

`M-x set-variable RET` *var* `RET` *value* `RET`
: Change the value of variable *var* to *value*.

To examine the value of a variable, use `C-h v` (`describe-variable`). This reads a variable name using the minibuffer, with completion, and displays both the value and the documentation of the variable. For example,

```
C-h v fill-column RET
```

displays something like this:

```
fill-column is a variable defined in ‘C source code’.
fill-column’s value is 70

Automatically becomes buffer-local when set.
This variable is safe as a file local variable if its value
satisfies the predicate ‘integerp’.

Documentation:
Column beyond which automatic line-wrapping should happen.
Interactively, you can set the local value with C-x f.

You can customize this variable.
```

The line that says "You can customize the variable" indicates that this variable is a user option. `C-h v` is not restricted to user options; it allows non-customizable variables too.

The most convenient way to set a specific customizable variable is with `M-x set-variable`. This reads the variable name with the minibuffer (with completion), and then reads a Lisp expression for the new value using the minibuffer a second time (you can insert the old value into the minibuffer for editing via `M-n`). For example,

```
M-x set-variable RET fill-column RET 75 RET
```

sets `fill-column` to 75.

`M-x set-variable` is limited to customizable variables, but you can set any variable with a Lisp expression like this:

```
(setq fill-column 75)
```

To execute such an expression, type `M-:` (`eval-expression`) and enter the expression in the minibuffer (see Section 24.9 [Lisp Eval], page 550). Alternatively, go to the '`*scratch*`' buffer, type in the expression, and then type `C-j` (see Section 24.10 [Lisp Interaction], page 552).

Setting variables, like all means of customizing Emacs except where otherwise stated, affects only the current Emacs session. The only way to alter the variable in future sessions is to put something in your initialization file (see Section 33.4 [Init File], page 711).

### 33.2.2 Hooks

*Hooks* are an important mechanism for customizing Emacs. A hook is a Lisp variable which holds a list of functions, to be called on some well-defined occasion. (This is called *running the hook*.) The individual functions in the list are called the *hook functions* of the hook. For example, the hook `kill-emacs-hook` runs just before exiting Emacs (see Section 3.2 [Exiting], page 57).

Most hooks are *normal hooks*. This means that when Emacs runs the hook, it calls each hook function in turn, with no arguments. We have made an effort to keep most hooks normal, so that you can use them in a uniform way. Every variable whose name ends in '`-hook`' is a normal hook.

A few hooks are *abnormal hooks*. Their names end in '`-functions`', instead of '`-hook`' (some old code may also use the deprecated suffix '`-hooks`'). What makes these hooks abnormal is the way its functions are called—perhaps they are given arguments, or perhaps the values they return are used in some way. For example, `find-file-not-found-functions` is abnormal because as soon as one hook function returns a non-`nil` value, the rest are not called at all (see ⟨undefined⟩ [Visiting], page ⟨undefined⟩). The documentation of each abnormal hook variable explains how its functions are used.

You can set a hook variable with `setq` like any other Lisp variable, but the recommended way to add a function to a hook (either normal or abnormal) is to use `add-hook`, as shown by the following examples. See Section "Hooks" in *The Emacs Lisp Reference Manual*, for details.

Most major modes run one or more *mode hooks* as the last step of initialization. Mode hooks are a convenient way to customize the behavior of individual modes; they are always normal. For example, here's how to set up a hook to turn on Auto Fill mode in Text mode and other modes based on Text mode:

```
(add-hook 'text-mode-hook 'auto-fill-mode)
```

This works by calling `auto-fill-mode`, which enables the minor mode when no argument is supplied (see Section 20.3 [Minor Modes], page 413). Next, suppose you don't want Auto Fill mode turned on in LaTeX mode, which is one of the modes based on Text mode. You can do this with the following additional line:

```
(add-hook 'latex-mode-hook (lambda () (auto-fill-mode -1)))
```

Here we have used the special macro `lambda` to construct an anonymous function (see Section "Lambda Expressions" in *The Emacs Lisp Reference Manual*), which calls `auto-fill-mode` with an argument of `-1` to disable the minor mode. Because LaTeX mode runs `latex-mode-hook` after running `text-mode-hook`, the result leaves Auto Fill mode disabled.

Here is a more complex example, showing how to use a hook to customize the indentation of C code:

```
(setq my-c-style
  '((c-comment-only-line-offset . 4)
```

```
        (c-cleanup-list . (scope-operator
                           empty-defun-braces
                           defun-close-semi))))

    (add-hook 'c-mode-common-hook
      (lambda () (c-add-style "my-style" my-c-style t)))
```

Major mode hooks also apply to other major modes *derived* from the original mode (see Section "Derived Modes" in *The Emacs Lisp Reference Manual*). For instance, HTML mode is derived from Text mode (see ⟨undefined⟩ [HTML Mode], page ⟨undefined⟩); when HTML mode is enabled, it runs `text-mode-hook` before running `html-mode-hook`. This provides a convenient way to use a single hook to affect several related modes. In particular, if you want to apply a hook function to any programming language mode, add it to `prog-mode-hook`; Prog mode is a major mode that does little else than to let other major modes inherit from it, exactly for this purpose.

It is best to design your hook functions so that the order in which they are executed does not matter. Any dependence on the order is asking for trouble. However, the order is predictable: the hook functions are executed in the order they appear in the hook.

If you play with adding various different versions of a hook function by calling `add-hook` over and over, remember that all the versions you added will remain in the hook variable together. You can clear out individual functions by calling `remove-hook`, or do `(setq` *`hook-variable`* `nil)` to remove everything.

If the hook variable is buffer-local, the buffer-local variable will be used instead of the global variable. However, if the buffer-local variable contains the element `t`, the global hook variable will be run as well.

### 33.2.3 Local Variables

`M-x make-local-variable RET` *`var`* `RET`
: Make variable *var* have a local value in the current buffer.

`M-x kill-local-variable RET` *`var`* `RET`
: Make variable *var* use its global value in the current buffer.

`M-x make-variable-buffer-local RET` *`var`* `RET`
: Mark variable *var* so that setting it will make it local to the buffer that is current at that time.

Almost any variable can be made *local* to a specific Emacs buffer. This means that its value in that buffer is independent of its value in other buffers. A few variables are always local in every buffer. Every other Emacs variable has a *global* value which is in effect in all buffers that have not made the variable local.

`M-x make-local-variable` reads the name of a variable and makes it local to the current buffer. Changing its value subsequently in this buffer will not affect others, and changes in its global value will not affect this buffer.

`M-x make-variable-buffer-local` marks a variable so it will become local automatically whenever it is set. More precisely, once a variable has been marked in this way, the usual ways of setting the variable automatically do `make-local-variable` first. We call such variables *per-buffer* variables. Many variables in Emacs are normally per-buffer; the

variable's document string tells you when this is so. A per-buffer variable's global value is normally never effective in any buffer, but it still has a meaning: it is the initial value of the variable for each new buffer.

Major modes (see Section 20.2 [Major Modes], page 399) always make variables local to the buffer before setting the variables. This is why changing major modes in one buffer has no effect on other buffers. Minor modes also work by setting variables—normally, each minor mode has one controlling variable which is non-`nil` when the mode is enabled (see Section 20.3 [Minor Modes], page 413). For many minor modes, the controlling variable is per buffer, and thus always buffer-local. Otherwise, you can make it local in a specific buffer like any other variable.

A few variables cannot be local to a buffer because they are always local to each display instead (see ⟨undefined⟩ [Multiple Displays], page ⟨undefined⟩). If you try to make one of these variables buffer-local, you'll get an error message.

`M-x kill-local-variable` makes a specified variable cease to be local to the current buffer. The global value of the variable henceforth is in effect in this buffer. Setting the major mode kills all the local variables of the buffer except for a few variables specially marked as *permanent locals*.

To set the global value of a variable, regardless of whether the variable has a local value in the current buffer, you can use the Lisp construct `setq-default`. This construct is used just like `setq`, but it sets variables' global values instead of their local values (if any). When the current buffer does have a local value, the new global value may not be visible until you switch to another buffer. Here is an example:

```
(setq-default fill-column 75)
```

`setq-default` is the only way to set the global value of a variable that has been marked with `make-variable-buffer-local`.

Lisp programs can use `default-value` to look at a variable's default value. This function takes a symbol as argument and returns its default value. The argument is evaluated; usually you must quote it explicitly. For example, here's how to obtain the default value of `fill-column`:

```
(default-value 'fill-column)
```

### 33.2.4 Local Variables in Files

A file can specify local variable values to use when editing the file with Emacs. Visiting the file checks for local variable specifications; it automatically makes these variables local to the buffer, and sets them to the values specified in the file.

#### 33.2.4.1 Specifying File Variables

There are two ways to specify file local variable values: in the first line, or with a local variables list. Here's how to specify them in the first line:

```
-*- mode: modename; var: value; ... -*-
```

You can specify any number of variable/value pairs in this way, each pair with a colon and semicolon. The special variable/value pair `mode: modename;`, if present, specifies a major mode. The *value*s are used literally, and not evaluated.

You can use `M-x add-file-local-variable-prop-line` instead of adding entries by hand. This command prompts for a variable and value, and adds them to the

first line in the appropriate way. `M-x delete-file-local-variable-prop-line` prompts for a variable, and deletes its entry from the line. The command `M-x copy-dir-locals-to-file-locals-prop-line` copies the current directory-local variables to the first line (see Section 33.2.5 [Directory Variables], page 701).

Here is an example first line that specifies Lisp mode and sets two variables with numeric values:

```
;; -*- mode: Lisp; fill-column: 75; comment-column: 50; -*-
```

Aside from `mode`, other keywords that have special meanings as file variables are `coding`, `unibyte`, and `eval`. These are described below.

In shell scripts, the first line is used to identify the script interpreter, so you cannot put any local variables there. To accommodate this, Emacs looks for local variable specifications in the *second* line if the first line specifies an interpreter. The same is true for man pages which start with the magic string '`'\"`' to specify a list of troff preprocessors (not all do, however).

Apart from using a '`-*-`' line, you can define file local variables using a *local variables list* near the end of the file. The start of the local variables list should be no more than 3000 characters from the end of the file, and must be on the last page if the file is divided into pages.

If a file has both a local variables list and a '`-*-`' line, Emacs processes *everything* in the '`-*-`' line first, and *everything* in the local variables list afterward. The exception to this is a major mode specification. Emacs applies this first, wherever it appears, since most major modes kill all local variables as part of their initialization.

A local variables list starts with a line containing the string '`Local Variables:`', and ends with a line containing the string '`End:`'. In between come the variable names and values, one set per line, like this:

```
/* Local Variables:  */
/* mode: c           */
/* comment-column: 0 */
/* End:              */
```

In this example, each line starts with the prefix '`/*`' and ends with the suffix '`*/`'. Emacs recognizes the prefix and suffix by finding them surrounding the magic string '`Local Variables:`', on the first line of the list; it then automatically discards them from the other lines of the list. The usual reason for using a prefix and/or suffix is to embed the local variables list in a comment, so it won't confuse other programs that the file is intended for. The example above is for the C programming language, where comments start with '`/*`' and end with '`*/`'.

Instead of typing in the local variables list directly, you can use the command `M-x add-file-local-variable`. This prompts for a variable and value, and adds them to the list, adding the '`Local Variables:`' string and start and end markers as necessary. The command `M-x delete-file-local-variable` deletes a variable from the list. `M-x copy-dir-locals-to-file-locals` copies directory-local variables to the list (see Section 33.2.5 [Directory Variables], page 701).

As with the '`-*-`' line, the variables in a local variables list are used literally, and are not evaluated first. If you want to split a long string value across multiple lines of the file, you can use backslash-newline, which is ignored in Lisp string constants; you should put

the prefix and suffix on each line, even lines that start or end within the string, as they will be stripped off when processing the list. Here is an example:

```
# Local Variables:
# compile-command: "cc foo.c -Dfoo=bar -Dhack=whatever \
#   -Dmumble=blaah"
# End:
```

Some "variable names" have special meanings in a local variables list:

- `mode` enables the specified major mode.
- `eval` evaluates the specified Lisp expression (the value returned by that expression is ignored).
- `coding` specifies the coding system for character code conversion of this file. See Section 19.6 [Coding Systems], page 381.
- `unibyte` says to load or compile a file of Emacs Lisp in unibyte mode, if the value is `t`. See Section 19.2 [Disabling Multibyte], page 376.

These four keywords are not really variables; setting them in any other context has no special meaning.

Do not use the `mode` keyword for minor modes. To enable or disable a minor mode in a local variables list, use the `eval` keyword with a Lisp expression that runs the mode command (see Section 20.3 [Minor Modes], page 413). For example, the following local variables list enables Eldoc mode (see Section 23.6.3 [Lisp Doc], page 527) by calling `eldoc-mode` with no argument (calling it with an argument of 1 would do the same), and disables Font Lock mode (see ⟨undefined⟩ [Font Lock], page ⟨undefined⟩) by calling `font-lock-mode` with an argument of -1.

```
;; Local Variables:
;; eval: (eldoc-mode)
;; eval: (font-lock-mode -1)
;; End:
```

Note, however, that it is often a mistake to specify minor modes this way. Minor modes represent individual user preferences, and it may be inappropriate to impose your preferences on another user who might edit the file. If you wish to automatically enable or disable a minor mode in a situation-dependent way, it is often better to do it in a major mode hook (see Section 33.2.2 [Hooks], page 696).

Use the command `M-x normal-mode` to reset the local variables and major mode of a buffer according to the file name and contents, including the local variables list if any. See ⟨undefined⟩ [Choosing Modes], page ⟨undefined⟩.

### 33.2.4.2 Safety of File Variables

File-local variables can be dangerous; when you visit someone else's file, there's no telling what its local variables list could do to your Emacs. Improper values of the `eval` "variable", and other variables such as `load-path`, could execute Lisp code you didn't intend to run.

Therefore, whenever Emacs encounters file local variable values that are not known to be safe, it displays the file's entire local variables list, and asks you for confirmation before setting them. You can type `y` or SPC to put the local variables list into effect, or `n` to ignore

it. When Emacs is run in batch mode (see Section E.2 [Initial Options], page 755), it can't really ask you, so it assumes the answer `n`.

Emacs normally recognizes certain variable/value pairs as safe. For instance, it is safe to give `comment-column` or `fill-column` any integer value. If a file specifies only known-safe variable/value pairs, Emacs does not ask for confirmation before setting them. Otherwise, you can tell Emacs to record all the variable/value pairs in this file as safe, by typing `!` at the confirmation prompt. When Emacs encounters these variable/value pairs subsequently, in the same file or others, it will assume they are safe.

Some variables, such as `load-path`, are considered particularly *risky*: there is seldom any reason to specify them as local variables, and changing them can be dangerous. If a file contains only risky local variables, Emacs neither offers nor accepts `!` as input at the confirmation prompt. If some of the local variables in a file are risky, and some are only potentially unsafe, you can enter `!` at the prompt. It applies all the variables, but only marks the non-risky ones as safe for the future. If you really want to record safe values for risky variables, do it directly by customizing '`safe-local-variable-values`' (see Section 33.1 [Easy Customization], page 686).

The variable `enable-local-variables` allows you to change the way Emacs processes local variables. Its default value is `t`, which specifies the behavior described above. If it is `nil`, Emacs simply ignores all file local variables. `:safe` means use only the safe values and ignore the rest. Any other value says to query you about each file that has local variables, without trying to determine whether the values are known to be safe.

The variable `enable-local-eval` controls whether Emacs processes `eval` variables. The three possibilities for the variable's value are `t`, `nil`, and anything else, just as for `enable-local-variables`. The default is `maybe`, which is neither `t` nor `nil`, so normally Emacs does ask for confirmation about processing `eval` variables.

As an exception, Emacs never asks for confirmation to evaluate any `eval` form if that form occurs within the variable `safe-local-eval-forms`.

### 33.2.5 Per-Directory Local Variables

Sometimes, you may wish to define the same set of local variables to all the files in a certain directory and its subdirectories, such as the directory tree of a large software project. This can be accomplished with *directory-local variables*.

The usual way to define directory-local variables is to put a file named '`.dir-locals.el`'[1] in a directory. Whenever Emacs visits any file in that directory or any of its subdirectories, it will apply the directory-local variables specified in '`.dir-locals.el`', as though they had been defined as file-local variables for that file (see Section 33.2.4 [File Variables], page 698). Emacs searches for '`.dir-locals.el`' starting in the directory of the visited file, and moving up the directory tree. To avoid slowdown, this search is skipped for remote files. If needed, the search can be extended for remote files by setting the variable `enable-remote-dir-locals` to `t`.

The '`.dir-locals.el`' file should hold a specially-constructed list, which maps major mode names (symbols) to alists (see Section "Association Lists" in *The Emacs Lisp Refer-*

[1] On MS-DOS, the name of this file should be '`_dir-locals.el`', due to limitations of the DOS filesystems. If the filesystem is limited to 8+3 file names, the name of the file will be truncated by the OS to '`_dir-loc.el`'.

*ence Manual*). Each alist entry consists of a variable name and the directory-local value to assign to that variable, when the specified major mode is enabled. Instead of a mode name, you can specify '`nil`', which means that the alist applies to any mode; or you can specify a subdirectory name (a string), in which case the alist applies to all files in that subdirectory.

Here's an example of a '`.dir-locals.el`' file:

```
((nil . ((indent-tabs-mode . t)
         (fill-column . 80)))
 (c-mode . ((c-file-style . "BSD")
            (subdirs . nil)))
 ("src/imported"
  . ((nil . ((change-log-default-name
              . "ChangeLog.local"))))))
```

This sets '`indent-tabs-mode`' and `fill-column` for any file in the directory tree, and the indentation style for any C source file. The special `subdirs` element is not a variable, but a special keyword which indicates that the C mode settings are only to be applied in the current directory, not in any subdirectories. Finally, it specifies a different '`ChangeLog`' file name for any file in the '`src/imported`' subdirectory.

Instead of editing the '`.dir-locals.el`' file by hand, you can use the command `M-x add-dir-local-variable`. This prompts for a mode or subdirectory name, and for variable and value, and adds the entry defining the directory-local variable. `M-x delete-dir-local-variable` deletes an entry. `M-x copy-file-locals-to-dir-locals` copies the file-local variables in the current file into '`.dir-locals.el`'.

Another method of specifying directory-local variables is to define a group of variables/value pairs in a *directory class*, using the `dir-locals-set-class-variables` function; then, tell Emacs which directories correspond to the class by using the `dir-locals-set-directory-class` function. These function calls normally go in your initialization file (see Section 33.4 [Init File], page 711). This method is useful when you can't put '`.dir-locals.el`' in a directory for some reason. For example, you could apply settings to an unwritable directory this way:

```
(dir-locals-set-class-variables 'unwritable-directory
   '((nil . ((some-useful-setting . value)))))

(dir-locals-set-directory-class
   "/usr/include/" 'unwritable-directory)
```

If a variable has both a directory-local and file-local value specified, the file-local value takes effect. Unsafe directory-local variables are handled in the same way as unsafe file-local variables (see Section 33.2.4.2 [Safe File Variables], page 700).

Directory-local variables also take effect in certain buffers that do not visit a file directly but perform work within a directory, such as Dired buffers (see Chapter 27 [Dired], page 588).

## 33.3 Customizing Key Bindings

This section describes *key bindings*, which map keys to commands, and *keymaps*, which record key bindings. It also explains how to customize key bindings, which is done by editing your init file (see Section 33.3.6 [Init Rebinding], page 706).

### 33.3.1 Keymaps

As described in ⟨undefined⟩ [Commands], page ⟨undefined⟩, each Emacs command is a Lisp function whose definition provides for interactive use. Like every Lisp function, a command has a function name, which usually consists of lower-case letters and hyphens.

A *key sequence* (*key*, for short) is a sequence of *input events* that have a meaning as a unit. Input events include characters, function keys and mouse buttons—all the inputs that you can send to the computer. A key sequence gets its meaning from its *binding*, which says what command it runs.

The bindings between key sequences and command functions are recorded in data structures called *keymaps*. Emacs has many of these, each used on particular occasions.

The *global* keymap is the most important keymap because it is always in effect. The global keymap defines keys for Fundamental mode (see Section 20.2 [Major Modes], page 399); most of these definitions are common to most or all major modes. Each major or minor mode can have its own keymap which overrides the global definitions of some keys.

For example, a self-inserting character such as `g` is self-inserting because the global keymap binds it to the command `self-insert-command`. The standard Emacs editing characters such as `C-a` also get their standard meanings from the global keymap. Commands to rebind keys, such as `M-x global-set-key`, work by storing the new binding in the proper place in the global map (see Section 33.3.5 [Rebinding], page 705).

Most modern keyboards have function keys as well as character keys. Function keys send input events just as character keys do, and keymaps can have bindings for them. Key sequences can mix function keys and characters. For example, if your keyboard has a HOME function key, Emacs can recognize key sequences like `C-x HOME`. You can even mix mouse events with keyboard events, such as `S-down-mouse-1`.

On text terminals, typing a function key actually sends the computer a sequence of characters; the precise details of the sequence depends on the function key and on the terminal type. (Often the sequence starts with `ESC [`.) If Emacs understands your terminal type properly, it automatically handles such sequences as single input events.

### 33.3.2 Prefix Keymaps

Internally, Emacs records only single events in each keymap. Interpreting a key sequence of multiple events involves a chain of keymaps: the first keymap gives a definition for the first event, which is another keymap, which is used to look up the second event in the sequence, and so on. Thus, a prefix key such as `C-x` or ESC has its own keymap, which holds the definition for the event that immediately follows that prefix.

The definition of a prefix key is usually the keymap to use for looking up the following event. The definition can also be a Lisp symbol whose function definition is the following keymap; the effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of `C-x` is the symbol `Control-X-prefix`, whose function definition is the keymap for `C-x` commands. The definitions of `C-c`, `C-x`, `C-h` and ESC as prefix keys appear in the global map, so these prefix keys are always available.

Aside from ordinary prefix keys, there is a fictitious "prefix key" which represents the menu bar; see Section "Menu Bar" in *The Emacs Lisp Reference Manual*, for special infor-

mation about menu bar key bindings. Mouse button events that invoke pop-up menus are also prefix keys; see Section "Menu Keymaps" in *The Emacs Lisp Reference Manual*, for more details.

Some prefix keymaps are stored in variables with names:

- `ctl-x-map` is the variable name for the map used for characters that follow `C-x`.
- `help-map` is for characters that follow `C-h`.
- `esc-map` is for characters that follow ESC. Thus, all Meta characters are actually defined by this map.
- `ctl-x-4-map` is for characters that follow `C-x 4`.
- `mode-specific-map` is for characters that follow `C-c`.

### 33.3.3 Local Keymaps

So far, we have explained the ins and outs of the global map. Major modes customize Emacs by providing their own key bindings in *local keymaps*. For example, C mode overrides TAB to make it indent the current line for C code. Minor modes can also have local keymaps; whenever a minor mode is in effect, the definitions in its keymap override both the major mode's local keymap and the global keymap. In addition, portions of text in the buffer can specify their own keymaps, which override all other keymaps.

A local keymap can redefine a key as a prefix key by defining it as a prefix keymap. If the key is also defined globally as a prefix, its local and global definitions (both keymaps) effectively combine: both definitions are used to look up the event that follows the prefix key. For example, if a local keymap defines `C-c` as a prefix keymap, and that keymap defines `C-z` as a command, this provides a local meaning for `C-c C-z`. This does not affect other sequences that start with `C-c`; if those sequences don't have their own local bindings, their global bindings remain in effect.

Another way to think of this is that Emacs handles a multi-event key sequence by looking in several keymaps, one by one, for a binding of the whole key sequence. First it checks the minor mode keymaps for minor modes that are enabled, then it checks the major mode's keymap, and then it checks the global keymap. This is not precisely how key lookup works, but it's good enough for understanding the results in ordinary circumstances.

### 33.3.4 Minibuffer Keymaps

The minibuffer has its own set of local keymaps; they contain various completion and exit commands.

- `minibuffer-local-map` is used for ordinary input (no completion).
- `minibuffer-local-ns-map` is similar, except that SPC exits just like RET.
- `minibuffer-local-completion-map` is for permissive completion.
- `minibuffer-local-must-match-map` is for strict completion and for cautious completion.
- `minibuffer-local-filename-completion-map` and `minibuffer-local-filename-must-match-map` are like the two previous ones, but they are specifically for file name completion. They do not bind SPC.

### 33.3.5 Changing Key Bindings Interactively

The way to redefine an Emacs key is to change its entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local bindings for the same key). Or you can change a local keymap, which affects all buffers using the same major mode.

In this section, we describe how to rebind keys for the present Emacs session. See Section 33.3.6 [Init Rebinding], page 706, for a description of how to make key rebindings affect future Emacs sessions.

`M-x global-set-key RET` *key cmd* `RET`
: Define *key* globally to run *cmd*.

`M-x local-set-key RET` *key cmd* `RET`
: Define *key* locally (in the major mode now in effect) to run *cmd*.

`M-x global-unset-key RET` *key*
: Make *key* undefined in the global map.

`M-x local-unset-key RET` *key*
: Make *key* undefined locally (in the major mode now in effect).

For example, the following binds `C-z` to the `shell` command (see Section 31.3.2 [Interactive Shell], page 656), replacing the normal global definition of `C-z`:

```
M-x global-set-key RET C-z shell RET
```

The `global-set-key` command reads the command name after the key. After you press the key, a message like this appears so that you can confirm that you are binding the key you want:

```
Set key C-z to command:
```

You can redefine function keys and mouse events in the same way; just type the function key or click the mouse when it's time to specify the key to rebind.

You can rebind a key that contains more than one event in the same way. Emacs keeps reading the key to rebind until it is a complete key (that is, not a prefix key). Thus, if you type `C-f` for *key*, that's the end; it enters the minibuffer immediately to read *cmd*. But if you type `C-x`, since that's a prefix, it reads another character; if that is `4`, another prefix character, it reads one more character, and so on. For example,

```
M-x global-set-key RET C-x 4 $ spell-other-window RET
```

redefines `C-x 4 $` to run the (fictitious) command `spell-other-window`.

You can remove the global definition of a key with `global-unset-key`. This makes the key *undefined*; if you type it, Emacs will just beep. Similarly, `local-unset-key` makes a key undefined in the current major mode keymap, which makes the global definition (or lack of one) come back into effect in that major mode.

If you have redefined (or undefined) a key and you subsequently wish to retract the change, undefining the key will not do the job—you need to redefine the key with its standard definition. To find the name of the standard definition of a key, go to a Fundamental mode buffer in a fresh Emacs and use `C-h c`. The documentation of keys in this manual also lists their command names.

If you want to prevent yourself from invoking a command by mistake, it is better to disable the command than to undefine the key. A disabled command is less work to invoke when you really want to. See Section 33.3.11 [Disabling], page 711.

### 33.3.6 Rebinding Keys in Your Init File

If you have a set of key bindings that you like to use all the time, you can specify them in your initialization file by writing Lisp code. See Section 33.4 [Init File], page 711, for a description of the initialization file.

There are several ways to write a key binding using Lisp. The simplest is to use the `kbd` function, which converts a textual representation of a key sequence—similar to how we have written key sequences in this manual—into a form that can be passed as an argument to `global-set-key`. For example, here's how to bind `C-z` to the `shell` command (see Section 31.3.2 [Interactive Shell], page 656):

```
(global-set-key (kbd "C-z") 'shell)
```

The single-quote before the command name, `shell`, marks it as a constant symbol rather than a variable. If you omit the quote, Emacs would try to evaluate `shell` as a variable. This probably causes an error; it certainly isn't what you want.

Here are some additional examples, including binding function keys and mouse events:

```
(global-set-key (kbd "C-c y") 'clipboard-yank)
(global-set-key (kbd "C-M-q") 'query-replace)
(global-set-key (kbd "<f5>") 'flyspell-mode)
(global-set-key (kbd "C-<f5>") 'linum-mode)
(global-set-key (kbd "C-<right>") 'forward-sentence)
(global-set-key (kbd "<mouse-2>") 'mouse-save-then-kill)
```

Instead of using `kbd`, you can use a Lisp string or vector to specify the key sequence. Using a string is simpler, but only works for ASCII characters and Meta-modified ASCII characters. For example, here's how to bind `C-x M-l` to `make-symbolic-link` (see ⟨undefined⟩ [Misc File Ops], page ⟨undefined⟩):

```
(global-set-key "\C-x\M-l" 'make-symbolic-link)
```

To put TAB, RET, ESC, or DEL in the string, use the Emacs Lisp escape sequences '`\t`', '`\r`', '`\e`', and '`\d`' respectively. Here is an example which binds `C-x TAB` to `indent-rigidly` (see Section 22.17 [Indentation], page 483):

```
(global-set-key "\C-x\t" 'indent-rigidly)
```

When the key sequence includes function keys or mouse button events, or non-ASCII characters such as `C-=` or `H-a`, you can use a vector to specify the key sequence. Each element in the vector stands for an input event; the elements are separated by spaces and surrounded by a pair of square brackets. If a vector element is a character, write it as a Lisp character constant: '`?`' followed by the character as it would appear in a string. Function keys are represented by symbols (see Section 33.3.8 [Function Keys], page 707); simply write the symbol's name, with no other delimiters or punctuation. Here are some examples:

```
(global-set-key [?\C-=] 'make-symbolic-link)
(global-set-key [?\M-\C-=] 'make-symbolic-link)
(global-set-key [?\H-a] 'make-symbolic-link)
(global-set-key [f7] 'make-symbolic-link)
```

```
(global-set-key [C-mouse-1] 'make-symbolic-link)
```

You can use a vector for the simple cases too:

```
(global-set-key [?\C-z ?\M-l] 'make-symbolic-link)
```

Language and coding systems may cause problems with key bindings for non-ASCII characters. See Section 33.4.5 [Init Non-ASCII], page 716.

As described in Section 33.3.3 [Local Keymaps], page 704, major modes and minor modes can define local keymaps. These keymaps are constructed when the mode is used for the first time in a session. If you wish to change one of these keymaps, you must use the *mode hook* (see Section 33.2.2 [Hooks], page 696).

For example, Texinfo mode runs the hook `texinfo-mode-hook`. Here's how you can use the hook to add local bindings for `C-c n` and `C-c p` in Texinfo mode:

```
(add-hook 'texinfo-mode-hook
          (lambda ()
            (define-key texinfo-mode-map "\C-cp"
                        'backward-paragraph)
            (define-key texinfo-mode-map "\C-cn"
                        'forward-paragraph)))
```

### 33.3.7 Modifier Keys

The default key bindings in Emacs are set up so that modified alphabetical characters are case-insensitive. In other words, `C-A` does the same thing as `C-a`, and `M-A` does the same thing as `M-a`. This concerns only alphabetical characters, and does not apply to "shifted" versions of other keys; for instance, `C-@` is not the same as `C-2`.

A CONTROL-modified alphabetical character is always considered case-insensitive: Emacs always treats `C-A` as `C-a`, `C-B` as `C-b`, and so forth. The reason for this is historical.

For all other modifiers, you can make the modified alphabetical characters case-sensitive when you customize Emacs. For instance, you could make `M-a` and `M-A` run different commands.

Although only the CONTROL and META modifier keys are commonly used, Emacs supports three other modifier keys. These are called SUPER, HYPER and ALT. Few terminals provide ways to use these modifiers; the key labeled ALT on most keyboards usually issues the META modifier, not ALT. The standard key bindings in Emacs do not include any characters with these modifiers. However, you can customize Emacs to assign meanings to them. The modifier bits are labeled as '`s-`', '`H-`' and '`A-`' respectively.

Even if your keyboard lacks these additional modifier keys, you can enter it using `C-x @`: `C-x @ h` adds the "hyper" flag to the next character, `C-x @ s` adds the "super" flag, and `C-x @ a` adds the "alt" flag. For instance, `C-x @ h C-a` is a way to enter `Hyper-Control-a`. (Unfortunately, there is no way to add two modifiers by using `C-x @` twice for the same character, because the first one goes to work on the `C-x`.)

### 33.3.8 Rebinding Function Keys

Key sequences can contain function keys as well as ordinary characters. Just as Lisp characters (actually integers) represent keyboard characters, Lisp symbols represent function keys. If the function key has a word as its label, then that word is also the name of the

corresponding Lisp symbol. Here are the conventional Lisp names for common function keys:

`left`, `up`, `right`, `down`
: Cursor arrow keys.

`begin`, `end`, `home`, `next`, `prior`
: Other cursor repositioning keys.

`select`, `print`, `execute`, `backtab`
`insert`, `undo`, `redo`, `clearline`
`insertline`, `deleteline`, `insertchar`, `deletechar`
: Miscellaneous function keys.

`f1`, `f2`, `...` `f35`
: Numbered function keys (across the top of the keyboard).

`kp-add`, `kp-subtract`, `kp-multiply`, `kp-divide`
`kp-backtab`, `kp-space`, `kp-tab`, `kp-enter`
`kp-separator`, `kp-decimal`, `kp-equal`
: Keypad keys (to the right of the regular keyboard), with names or punctuation.

`kp-0`, `kp-1`, `...` `kp-9`
: Keypad keys with digits.

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`
: Keypad PF keys.

These names are conventional, but some systems (especially when using X) may use different names. To make certain what symbol is used for a given function key on your terminal, type `C-h c` followed by that key.

See Section 33.3.6 [Init Rebinding], page 706, for examples of binding function keys.

Many keyboards have a "numeric keypad" on the right hand side. The numeric keys in the keypad double up as cursor motion keys, toggled by a key labeled '`Num Lock`'. By default, Emacs translates these keys to the corresponding keys in the main keyboard. For example, when '`Num Lock`' is on, the key labeled '`8`' on the numeric keypad produces `kp-8`, which is translated to `8`; when '`Num Lock`' is off, the same key produces `kp-up`, which is translated to UP. If you rebind a key such as `8` or UP, it affects the equivalent keypad key too. However, if you rebind a '`kp-`' key directly, that won't affect its non-keypad equivalent. Note that the modified keys are not translated: for instance, if you hold down the META key while pressing the '`8`' key on the numeric keypad, that generates `M-KP-8`.

Emacs provides a convenient method for binding the numeric keypad keys, using the variables `keypad-setup`, `keypad-numlock-setup`, `keypad-shifted-setup`, and `keypad-numlock-shifted-setup`. These can be found in the '`keyboard`' customization group (see Section 33.1 [Easy Customization], page 686). You can rebind the keys to perform other tasks, such as issuing numeric prefix arguments.

### 33.3.9 Named ASCII Control Characters

TAB, RET, BS, LFD, ESC and DEL started out as names for certain ASCII control characters, used so often that they have special keys of their own. For instance, TAB was another

name for `C-i`. Later, users found it convenient to distinguish in Emacs between these keys and the "same" control characters typed with the CTRL key. Therefore, on most modern terminals, they are no longer the same: TAB is different from `C-i`.

Emacs can distinguish these two kinds of input if the keyboard does. It treats the "special" keys as function keys named `tab`, `return`, `backspace`, `linefeed`, `escape`, and `delete`. These function keys translate automatically into the corresponding ASCII characters *if* they have no bindings of their own. As a result, neither users nor Lisp programs need to pay attention to the distinction unless they care to.

If you do not want to distinguish between (for example) TAB and `C-i`, make just one binding, for the ASCII character TAB (octal code 011). If you do want to distinguish, make one binding for this ASCII character, and another for the "function key" `tab`.

With an ordinary ASCII terminal, there is no way to distinguish between TAB and `C-i` (and likewise for other such pairs), because the terminal sends the same character in both cases.

### 33.3.10 Rebinding Mouse Buttons

Emacs uses Lisp symbols to designate mouse buttons, too. The ordinary mouse events in Emacs are *click* events; these happen when you press a button and release it without moving the mouse. You can also get *drag* events, when you move the mouse while holding the button down. Drag events happen when you finally let go of the button.

The symbols for basic click events are `mouse-1` for the leftmost button, `mouse-2` for the next, and so on. Here is how you can redefine the second mouse button to split the current window:

```
(global-set-key [mouse-2] 'split-window-below)
```

The symbols for drag events are similar, but have the prefix '`drag-`' before the word '`mouse`'. For example, dragging the first button generates a `drag-mouse-1` event.

You can also define bindings for events that occur when a mouse button is pressed down. These events start with '`down-`' instead of '`drag-`'. Such events are generated only if they have key bindings. When you get a button-down event, a corresponding click or drag event will always follow.

If you wish, you can distinguish single, double, and triple clicks. A double click means clicking a mouse button twice in approximately the same place. The first click generates an ordinary click event. The second click, if it comes soon enough, generates a double-click event instead. The event type for a double-click event starts with '`double-`': for example, `double-mouse-3`.

This means that you can give a special meaning to the second click at the same place, but it must act on the assumption that the ordinary single click definition has run when the first click was received.

This constrains what you can do with double clicks, but user interface designers say that this constraint ought to be followed in any case. A double click should do something similar to the single click, only "more so". The command for the double-click event should perform the extra work for the double click.

If a double-click event has no binding, it changes to the corresponding single-click event. Thus, if you don't define a particular double click specially, it executes the single-click command twice.

Emacs also supports triple-click events whose names start with '`triple-`'. Emacs does not distinguish quadruple clicks as event types; clicks beyond the third generate additional triple-click events. However, the full number of clicks is recorded in the event list, so if you know Emacs Lisp you can distinguish if you really want to (see Section "Click Events" in *The Emacs Lisp Reference Manual*). We don't recommend distinct meanings for more than three clicks, but sometimes it is useful for subsequent clicks to cycle through the same set of three meanings, so that four clicks are equivalent to one click, five are equivalent to two, and six are equivalent to three.

Emacs also records multiple presses in drag and button-down events. For example, when you press a button twice, then move the mouse while holding the button, Emacs gets a '`double-drag-`' event. And at the moment when you press it down for the second time, Emacs gets a '`double-down-`' event (which is ignored, like all button-down events, if it has no binding).

The variable `double-click-time` specifies how much time can elapse between clicks and still allow them to be grouped as a multiple click. Its value is in units of milliseconds. If the value is `nil`, double clicks are not detected at all. If the value is `t`, then there is no time limit. The default is 500.

The variable `double-click-fuzz` specifies how much the mouse can move between clicks and still allow them to be grouped as a multiple click. Its value is in units of pixels on windowed displays and in units of 1/8 of a character cell on text-mode terminals; the default is 3.

The symbols for mouse events also indicate the status of the modifier keys, with the usual prefixes '`C-`', '`M-`', '`H-`', '`s-`', '`A-`' and '`S-`'. These always precede '`double-`' or '`triple-`', which always precede '`drag-`' or '`down-`'.

A frame includes areas that don't show text from the buffer, such as the mode line and the scroll bar. You can tell whether a mouse button comes from a special area of the screen by means of dummy "prefix keys". For example, if you click the mouse in the mode line, you get the prefix key `mode-line` before the ordinary mouse-button symbol. Thus, here is how to define the command for clicking the first button in a mode line to run `scroll-up-command`:

```
(global-set-key [mode-line mouse-1] 'scroll-up-command)
```

Here is the complete list of these dummy prefix keys and their meanings:

`mode-line`
: The mouse was in the mode line of a window.

`vertical-line`
: The mouse was in the vertical line separating side-by-side windows. (If you use scroll bars, they appear in place of these vertical lines.)

`vertical-scroll-bar`
: The mouse was in a vertical scroll bar. (This is the only kind of scroll bar Emacs currently supports.)

`menu-bar`
: The mouse was in the menu bar.

`header-line`
: The mouse was in a header line.

You can put more than one mouse button in a key sequence, but it isn't usual to do so.

### 33.3.11 Disabling Commands

Disabling a command means that invoking it interactively asks for confirmation from the user. The purpose of disabling a command is to prevent users from executing it by accident; we do this for commands that might be confusing to the uninitiated.

Attempting to invoke a disabled command interactively in Emacs displays a window containing the command's name, its documentation, and some instructions on what to do immediately; then Emacs asks for input saying whether to execute the command as requested, enable it and execute it, or cancel. If you decide to enable the command, you must then answer another question—whether to do this permanently, or just for the current session. (Enabling permanently works by automatically editing your initialization file.) You can also type `!` to enable *all* commands, for the current session only.

The direct mechanism for disabling a command is to put a non-`nil` `disabled` property on the Lisp symbol for the command. Here is the Lisp program to do this:

```
(put 'delete-region 'disabled t)
```

If the value of the `disabled` property is a string, that string is included in the message displayed when the command is used:

```
(put 'delete-region 'disabled
     "It's better to use `kill-region' instead.\n")
```

You can make a command disabled either by editing the initialization file directly, or with the command `M-x disable-command`, which edits the initialization file for you. Likewise, `M-x enable-command` edits the initialization file to enable a command permanently. See Section 33.4 [Init File], page 711.

If Emacs was invoked with the '`-q`' or '`--no-init-file`' options (see Section E.2 [Initial Options], page 755), it will not edit your initialization file. Doing so could lose information because Emacs has not read your initialization file.

Whether a command is disabled is independent of what key is used to invoke it; disabling also applies if the command is invoked using `M-x`. However, disabling a command has no effect on calling it as a function from Lisp programs.

## 33.4 The Emacs Initialization File

When Emacs is started, it normally tries to load a Lisp program from an *initialization file*, or *init file* for short. This file, if it exists, specifies how to initialize Emacs for you. Emacs looks for your init file using the filenames '`~/.emacs`', '`~/.emacs.el`', or '`~/.emacs.d/init.el`'; you can choose to use any one of these three names (see Section 33.4.4 [Find Init], page 716). Here, '`~/`' stands for your home directory.

You can use the command line switch '`-q`' to prevent loading your init file, and '`-u`' (or '`--user`') to specify a different user's init file (see Section E.2 [Initial Options], page 755).

There can also be a *default init file*, which is the library named '`default.el`', found via the standard search path for libraries. The Emacs distribution contains no such library; your site may create one for local customizations. If this library exists, it is loaded whenever you start Emacs (except when you specify '`-q`'). But your init file, if any, is loaded first; if it sets `inhibit-default-init` non-`nil`, then '`default`' is not loaded.

Your site may also have a *site startup file*; this is named '`site-start.el`', if it exists. Like '`default.el`', Emacs finds this file via the standard search path for Lisp libraries.

Emacs loads this library before it loads your init file. To inhibit loading of this library, use the option '`--no-site-file`'. See Section E.2 [Initial Options], page 755. We recommend against using '`site-start.el`' for changes that some users may not like. It is better to put them in '`default.el`', so that users can more easily override them.

You can place '`default.el`' and '`site-start.el`' in any of the directories which Emacs searches for Lisp libraries. The variable `load-path` (see Section 24.8 [Lisp Libraries], page 549) specifies these directories. Many sites put these files in a subdirectory named '`site-lisp`' in the Emacs installation directory, such as '`/usr/local/share/emacs/site-lisp`'.

Byte-compiling your init file is not recommended (see Section "Byte Compilation" in *the Emacs Lisp Reference Manual*). It generally does not speed up startup very much, and often leads to problems when you forget to recompile the file. A better solution is to use the Emacs server to reduce the number of times you have to start Emacs (see Section 31.4 [Emacs Server], page 665). If your init file defines many functions, consider moving them to a separate (byte-compiled) file that you load in your init file.

If you are going to write actual Emacs Lisp programs that go beyond minor customization, you should read the *Emacs Lisp Reference Manual*.

### 33.4.1 Init File Syntax

The init file contains one or more Lisp expressions. Each of these consists of a function name followed by arguments, all surrounded by parentheses. For example, `(setq fill-column 60)` calls the function `setq` to set the variable `fill-column` (see Section 22.11 [Filling], page 473) to 60.

You can set any Lisp variable with `setq`, but with certain variables `setq` won't do what you probably want in the '`.emacs`' file. Some variables automatically become buffer-local when set with `setq`; what you want in '`.emacs`' is to set the default value, using `setq-default`. Some customizable minor mode variables do special things to enable the mode when you set them with Customize, but ordinary `setq` won't do that; to enable the mode in your '`.emacs`' file, call the minor mode command. The following section has examples of both of these methods.

The second argument to `setq` is an expression for the new value of the variable. This can be a constant, a variable, or a function call expression. In '`.emacs`', constants are used most of the time. They can be:

Numbers: Numbers are written in decimal, with an optional initial minus sign.

Strings: Lisp string syntax is the same as C string syntax with a few extra features. Use a double-quote character to begin and end a string constant.

In a string, you can include newlines and special characters literally. But often it is cleaner to use backslash sequences for them: '`\n`' for newline, '`\b`' for backspace, '`\r`' for carriage return, '`\t`' for tab, '`\f`' for formfeed (control-L), '`\e`' for escape, '`\\`' for a backslash, '`\"`' for a double-quote, or '`\ooo`' for the character whose octal code is *ooo*. Backslash and double-quote are the only characters for which backslash sequences are mandatory.

'`\C-`' can be used as a prefix for a control character, as in '`\C-s`' for ASCII control-S, and '`\M-`' can be used as a prefix for a Meta character, as in '`\M-a`' for `Meta-A` or '`\M-\C-a`' for `Control-Meta-A`.

See Section 33.4.5 [Init Non-ASCII], page 716, for information about including non-ASCII in your init file.

Characters:
: Lisp character constant syntax consists of a '`?`' followed by either a character or an escape sequence starting with '`\`'. Examples: `?x`, `?\n`, `?\"`, `?\)`. Note that strings and characters are not interchangeable in Lisp; some contexts require one and some contexts require the other.

  See Section 33.4.5 [Init Non-ASCII], page 716, for information about binding commands to keys which send non-ASCII characters.

True:
: `t` stands for 'true'.

False:
: `nil` stands for 'false'.

Other Lisp objects:
: Write a single-quote (`'`) followed by the Lisp object you want.

### 33.4.2 Init File Examples

Here are some examples of doing certain commonly desired things with Lisp expressions:

- Add a directory to the variable `load-path`. You can then put Lisp libraries that are not included with Emacs in this directory, and load them with `M-x load-library`. See Section 24.8 [Lisp Libraries], page 549.

  ```
  (add-to-list 'load-path "/path/to/lisp/libraries")
  ```

- Make TAB in C mode just insert a tab if point is in the middle of a line.

  ```
  (setq c-tab-always-indent nil)
  ```

  Here we have a variable whose value is normally `t` for 'true' and the alternative is `nil` for 'false'.

- Make searches case sensitive by default (in all buffers that do not override this).

  ```
  (setq-default case-fold-search nil)
  ```

  This sets the default value, which is effective in all buffers that do not have local values for the variable (see Section 33.2.3 [Locals], page 697). Setting `case-fold-search` with `setq` affects only the current buffer's local value, which is probably not what you want to do in an init file.

- Specify your own email address, if Emacs can't figure it out correctly.

  ```
  (setq user-mail-address "cheney@torture.gov")
  ```

  Various Emacs packages, such as Message mode, consult `user-mail-address` when they need to know your email address. See Section 29.2 [Mail Headers], page 625.

- Make Text mode the default mode for new buffers.

  ```
  (setq-default major-mode 'text-mode)
  ```

  Note that `text-mode` is used because it is the command for entering Text mode. The single-quote before it makes the symbol a constant; otherwise, `text-mode` would be treated as a variable name.

- Set up defaults for the Latin-1 character set which supports most of the languages of Western Europe.

```
(set-language-environment "Latin-1")
```

- Turn off Line Number mode, a global minor mode.

```
(line-number-mode 0)
```

- Turn on Auto Fill mode automatically in Text mode and related modes (see Section 33.2.2 [Hooks], page 696).

```
(add-hook 'text-mode-hook 'auto-fill-mode)
```

- Load the installed Lisp library named 'foo' (actually a file 'foo.elc' or 'foo.el' in a standard Emacs directory).

```
(load "foo")
```

  When the argument to load is a relative file name, not starting with '/' or '~', load searches the directories in load-path (see Section 24.8 [Lisp Libraries], page 549).

- Load the compiled Lisp file 'foo.elc' from your home directory.

```
(load "~/foo.elc")
```

  Here a full file name is used, so no searching is done.

- Tell Emacs to find the definition for the function myfunction by loading a Lisp library named 'mypackage' (i.e., a file 'mypackage.elc' or 'mypackage.el'):

```
(autoload 'myfunction "mypackage" "Do what I say." t)
```

  Here the string "Do what I say." is the function's documentation string. You specify it in the autoload definition so it will be available for help commands even when the package is not loaded. The last argument, t, indicates that this function is interactive; that is, it can be invoked interactively by typing M-x myfunction RET or by binding it to a key. If the function is not interactive, omit the t or use nil.

- Rebind the key C-x l to run the function make-symbolic-link (see Section 33.3.6 [Init Rebinding], page 706).

```
(global-set-key "\C-xl" 'make-symbolic-link)
```

  or

```
(define-key global-map "\C-xl" 'make-symbolic-link)
```

  Note once again the single-quote used to refer to the symbol make-symbolic-link instead of its value as a variable.

- Do the same thing for Lisp mode only.

```
(define-key lisp-mode-map "\C-xl" 'make-symbolic-link)
```

- Redefine all keys which now run next-line in Fundamental mode so that they run forward-line instead.

```
(substitute-key-definition 'next-line 'forward-line
                           global-map)
```

- Make C-x C-v undefined.

```
(global-unset-key "\C-x\C-v")
```

  One reason to undefine a key is so that you can make it a prefix. Simply defining C-x C-v *anything* will make C-x C-v a prefix, but C-x C-v must first be freed of its usual non-prefix definition.

- Make '$' have the syntax of punctuation in Text mode. Note the use of a character constant for '$'.

```
(modify-syntax-entry ?\$ "." text-mode-syntax-table)
```

- Enable the use of the command `narrow-to-region` without confirmation.

```
(put 'narrow-to-region 'disabled nil)
```

- Adjusting the configuration to various platforms and Emacs versions.

  Users typically want Emacs to behave the same on all systems, so the same init file is right for all platforms. However, sometimes it happens that a function you use for customizing Emacs is not available on some platforms or in older Emacs versions. To deal with that situation, put the customization inside a conditional that tests whether the function or facility is available, like this:

```
(if (fboundp 'blink-cursor-mode)
    (blink-cursor-mode 0))

(if (boundp 'coding-category-utf-8)
    (set-coding-priority '(coding-category-utf-8)))
```

  You can also simply disregard the errors that occur if the function is not defined.

```
(condition case ()
    (set-face-background 'region "grey75")
  (error nil))
```

  A `setq` on a variable which does not exist is generally harmless, so those do not need a conditional.

### 33.4.3 Terminal-specific Initialization

Each terminal type can have a Lisp library to be loaded into Emacs when it is run on that type of terminal. For a terminal type named *termtype*, the library is called '`term/`*`termtype`*' and it is found by searching the directories `load-path` as usual and trying the suffixes '`.elc`' and '`.el`'. Normally it appears in the subdirectory '`term`' of the directory where most Emacs libraries are kept.

The usual purpose of the terminal-specific library is to map the escape sequences used by the terminal's function keys onto more meaningful names, using `input-decode-map` (or `function-key-map` before it). See the file '`term/lk201.el`' for an example of how this is done. Many function keys are mapped automatically according to the information in the Termcap data base; the terminal-specific library needs to map only the function keys that Termcap does not specify.

When the terminal type contains a hyphen, only the part of the name before the first hyphen is significant in choosing the library name. Thus, terminal types '`aaa-48`' and '`aaa-30-rv`' both use the library '`term/aaa`'. The code in the library can use `(getenv "TERM")` to find the full terminal type name.

The library's name is constructed by concatenating the value of the variable `term-file-prefix` and the terminal type. Your '`.emacs`' file can prevent the loading of the terminal-specific library by setting `term-file-prefix` to `nil`.

Emacs runs the hook `term-setup-hook` at the end of initialization, after both your '`.emacs`' file and any terminal-specific library have been read in. Add hook functions to

this hook if you wish to override part of any of the terminal-specific libraries and to define initializations for terminals that do not have a library. See Section 33.2.2 [Hooks], page 696.

### 33.4.4 How Emacs Finds Your Init File

Normally Emacs uses the environment variable `HOME` (see Section E.4.1 [General Variables], page 758) to find '`.emacs`'; that's what '`~`' means in a file name. If '`.emacs`' is not found inside '`~/`' (nor '`.emacs.el`'), Emacs looks for '`~/.emacs.d/init.el`' (which, like '`~/.emacs.el`', can be byte-compiled).

However, if you run Emacs from a shell started by `su`, Emacs tries to find your own '`.emacs`', not that of the user you are currently pretending to be. The idea is that you should get your own editor customizations even if you are running as the super user.

More precisely, Emacs first determines which user's init file to use. It gets your user name from the environment variables `LOGNAME` and `USER`; if neither of those exists, it uses effective user-ID. If that user name matches the real user-ID, then Emacs uses `HOME`; otherwise, it looks up the home directory corresponding to that user name in the system's data base of users.

### 33.4.5 Non-ASCII Characters in Init Files

Language and coding systems may cause problems if your init file contains non-ASCII characters, such as accented letters, in strings or key bindings.

If you want to use non-ASCII characters in your init file, you should put a '`-*-coding:` *coding-system*`-*-`' tag on the first line of the init file, and specify a coding system that supports the character(s) in question. See Section 19.7 [Recognize Coding], page 383. This is because the defaults for decoding non-ASCII text might not yet be set up by the time Emacs reads those parts of your init file which use such strings, possibly leading Emacs to decode those strings incorrectly. You should then avoid adding Emacs Lisp code that modifies the coding system in other ways, such as calls to `set-language-environment`.

To bind non-ASCII keys, you must use a vector (see Section 33.3.6 [Init Rebinding], page 706). The string syntax cannot be used, since the non-ASCII characters will be interpreted as meta keys. For instance:

```
(global-set-key [?char] 'some-function)
```

Type `C-q`, followed by the key you want to bind, to insert *char*.

**Warning:** if you change the keyboard encoding, or change between multibyte and unibyte mode, or anything that would alter which code `C-q` would insert for that character, this key binding may stop working. It is therefore advisable to use one and only one coding system, for your init file as well as the files you edit. For example, don't mix the '`latin-1`' and '`latin-9`' coding systems.

# 34 Dealing with Common Problems

If you type an Emacs command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. Emacs bugs and system crashes are also considered.

## 34.1 Quitting and Aborting

`C-g`
`C-BREAK` (MS-DOS only)
: Quit: cancel running or partially typed command.

`C-]`
: Abort innermost recursive editing level and cancel the command which invoked it (`abort-recursive-edit`).

`ESC ESC ESC`
: Either quit or abort, whichever makes sense (`keyboard-escape-quit`).

`M-x top-level`
: Abort all recursive editing levels that are currently executing.

`C-/`
`C-x u`
`C-_`
: Cancel a previously made change in the buffer contents (`undo`).

There are two ways of canceling a command before it has finished: *quitting* with `C-g`, and *aborting* with `C-]` or `M-x top-level`. Quitting cancels a partially typed command, or one which is still running. Aborting exits a recursive editing level and cancels the command that invoked the recursive edit (see Section 31.9 [Recursive Edit], page 675).

Quitting with `C-g` is the way to get rid of a partially typed command, or a numeric argument that you don't want. Furthermore, if you are in the middle of a command that is running, `C-g` stops the command in a relatively safe way. For example, if you quit out of a kill command that is taking a long time, either your text will *all* still be in the buffer, or it will *all* be in the kill ring, or maybe both. If the region is active, `C-g` deactivates the mark, unless Transient Mark mode is off (see Section 8.7 [Disabled Transient Mark], page 94). If you are in the middle of an incremental search, `C-g` behaves specially; it may take two successive `C-g` characters to get out of a search. See Section 12.1 [Incremental Search], page 199, for details.

On MS-DOS, the character `C-BREAK` serves as a quit character like `C-g`. The reason is that it is not feasible, on MS-DOS, to recognize `C-g` while a command is running, between interactions with the user. By contrast, it *is* feasible to recognize `C-BREAK` at all times. See Section "MS-DOS Keyboard" in *Specialized Emacs Features*.

`C-g` works by setting the variable `quit-flag` to `t` the instant `C-g` is typed; Emacs Lisp checks this variable frequently, and quits if it is non-`nil`. `C-g` is only actually executed as a command if you type it while Emacs is waiting for input. In that case, the command it runs is `keyboard-quit`.

On a text terminal, if you quit with `C-g` a second time before the first `C-g` is recognized, you activate the "emergency escape" feature and return to the shell. See Section 34.2.8 [Emergency Escape], page 722.

There are some situations where you cannot quit. When Emacs is waiting for the operating system to do something, quitting is impossible unless special pains are taken for the particular system call within Emacs where the waiting occurs. We have done this for the system calls that users are likely to want to quit from, but it's possible you will encounter a case not handled. In one very common case—waiting for file input or output using NFS—Emacs itself knows how to quit, but many NFS implementations simply do not allow user programs to stop waiting for NFS when the NFS server is hung.

Aborting with `C-]` (`abort-recursive-edit`) is used to get out of a recursive editing level and cancel the command which invoked it. Quitting with `C-g` does not do this, and could not do this, because it is used to cancel a partially typed command *within* the recursive editing level. Both operations are useful. For example, if you are in a recursive edit and type `C-u 8` to enter a numeric argument, you can cancel that argument with `C-g` and remain in the recursive edit.

The sequence `ESC ESC ESC` (`keyboard-escape-quit`) can either quit or abort. (We defined it this way because ESC means "get out" in many PC programs.) It can cancel a prefix argument, clear a selected region, or get out of a Query Replace, like `C-g`. It can get out of the minibuffer or a recursive edit, like `C-]`. It can also get out of splitting the frame into multiple windows, as with `C-x 1`. One thing it cannot do, however, is stop a command that is running. That's because it executes as an ordinary command, and Emacs doesn't notice it until it is ready for the next command.

The command `M-x top-level` is equivalent to "enough" `C-]` commands to get you out of all the levels of recursive edits that you are in; it also exits the minibuffer if it is active. `C-]` gets you out one level at a time, but `M-x top-level` goes out all levels at once. Both `C-]` and `M-x top-level` are like all other commands, and unlike `C-g`, in that they take effect only when Emacs is ready for a command. `C-]` is an ordinary key and has its meaning only because of its binding in the keymap. See Section 31.9 [Recursive Edit], page 675.

`C-/` (`undo`) is not strictly speaking a way of canceling a command, but you can think of it as canceling a command that already finished executing. See Section 22.9 [Undo], page 469, for more information about the undo facility.

## 34.2 Dealing with Emacs Trouble

This section describes how to recognize and deal with situations in which Emacs does not work as you expect, such as keyboard code mixups, garbled displays, running out of memory, and crashes and hangs.

See Section 34.3 [Bugs], page 722, for what to do when you think you have found a bug in Emacs.

### 34.2.1 If DEL Fails to Delete

Every keyboard has a large key, usually labeled BACKSPACE, which is ordinarily used to erase the last character that you typed. In Emacs, this key is supposed to be equivalent to DEL.

When Emacs starts up on a graphical display, it determines automatically which key should be DEL. In some unusual cases, Emacs gets the wrong information from the system, and BACKSPACE ends up deleting forwards instead of backwards.

Some keyboards also have a DELETE key, which is ordinarily used to delete forwards. If this key deletes backward in Emacs, that too suggests Emacs got the wrong information—but in the opposite sense.

On a text terminal, if you find that BACKSPACE prompts for a Help command, like `Control-h`, instead of deleting a character, it means that key is actually sending the BS character. Emacs ought to be treating BS as DEL, but it isn't.

In all of those cases, the immediate remedy is the same: use the command `M-x normal-erase-is-backspace-mode`. This toggles between the two modes that Emacs supports for handling DEL, so if Emacs starts in the wrong mode, this should switch to the right mode. On a text terminal, if you want to ask for help when BS is treated as DEL, use F1; `C-?` may also work, if it sends character code 127.

To fix the problem in every Emacs session, put one of the following lines into your initialization file (see Section 33.4 [Init File], page 711). For the first case above, where BACKSPACE deletes forwards instead of backwards, use this line to make BACKSPACE act as DEL:

```
(normal-erase-is-backspace-mode 0)
```

For the other two cases, use this line:

```
(normal-erase-is-backspace-mode 1)
```

Another way to fix the problem for every Emacs session is to customize the variable `normal-erase-is-backspace`: the value `t` specifies the mode where BS or BACKSPACE is DEL, and `nil` specifies the other mode. See Section 33.1 [Easy Customization], page 686.

### 34.2.2 Recursive Editing Levels

Recursive editing levels are important and useful features of Emacs, but they can seem like malfunctions if you do not understand them.

If the mode line has square brackets '`[...]`' around the parentheses that contain the names of the major and minor modes, you have entered a recursive editing level. If you did not do this on purpose, or if you don't understand what that means, you should just get out of the recursive editing level. To do so, type `M-x top-level`. See Section 31.9 [Recursive Edit], page 675.

### 34.2.3 Garbage on the Screen

If the text on a text terminal looks wrong, the first thing to do is see whether it is wrong in the buffer. Type `C-l` to redisplay the entire screen. If the screen appears correct after this, the problem was entirely in the previous screen update. (Otherwise, see the following section.)

Display updating problems often result from an incorrect terminfo entry for the terminal you are using. The file '`etc/TERMS`' in the Emacs distribution gives the fixes for known problems of this sort. '`INSTALL`' contains general advice for these problems in one of its sections. If you seem to be using the right terminfo entry, it is possible that there is a bug in the terminfo entry, or a bug in Emacs that appears for certain terminal types.

### 34.2.4 Garbage in the Text

If `C-l` shows that the text is wrong, first type `C-h l` to see what commands you typed to produce the observed results. Then try undoing the changes step by step using `C-x u`, until it gets back to a state you consider correct.

If a large portion of text appears to be missing at the beginning or end of the buffer, check for the word '`Narrow`' in the mode line. If it appears, the text you don't see is probably still present, but temporarily off-limits. To make it accessible again, type `C-x n w`. See ⟨undefined⟩ [Narrowing], page ⟨undefined⟩.

### 34.2.5 Running out of Memory

If you get the error message '`Virtual memory exceeded`', save your modified buffers with `C-x s`. This method of saving them has the smallest need for additional memory. Emacs keeps a reserve of memory which it makes available when this error happens; that should be enough to enable `C-x s` to complete its work. When the reserve has been used, '`!MEM FULL!`' appears at the beginning of the mode line, indicating there is no more reserve.

Once you have saved your modified buffers, you can exit this Emacs session and start another, or you can use `M-x kill-some-buffers` to free space in the current Emacs job. If this frees up sufficient space, Emacs will refill its memory reserve, and '`!MEM FULL!`' will disappear from the mode line. That means you can safely go on editing in the same Emacs session.

Do not use `M-x buffer-menu` to save or kill buffers when you run out of memory, because the Buffer Menu needs a fair amount of memory itself, and the reserve supply may not be enough.

### 34.2.6 When Emacs Crashes

Emacs is not supposed to crash, but if it does, it produces a *crash report* prior to exiting. The crash report is printed to the standard error stream. If Emacs was started from a graphical desktop, the standard error stream is commonly redirected to a file such as '`~/.xsession-errors`', so you can look for the crash report there.

The format of the crash report depends on the platform. On some platforms, such as those using the GNU C Library, the crash report includes a *backtrace* describing the execution state prior to crashing, which can be used to help debug the crash. Here is an example:

```
Fatal error 11: Segmentation fault
Backtrace:
emacs[0x5094e4]
emacs[0x4ed3e6]
emacs[0x4ed504]
/lib64/libpthread.so.0[0x375220efe0]
/lib64/libpthread.so.0(read+0xe)[0x375220e08e]
emacs[0x509af6]
emacs[0x5acc26]
...
```

The number '`11`' is the system signal number corresponding to the crash—in this case a segmentation fault. The hexadecimal numbers are program addresses, which can be associ-

ated with source code lines using a debugging tool. For example, the GDB command '`list *0x509af6`' prints the source-code lines corresponding to the '`emacs[0x509af6]`' entry. If your system has the `addr2line` utility, the following shell command outputs a backtrace with source-code line numbers:

```
sed -n 's/.*\[\(.*\)]$/\1/p' backtrace |
  addr2line -C -f -i -e bindir/emacs
```

Here, *backtrace* is the name of a text file containing a copy of the backtrace, and *bindir* is the name of the directory that contains the Emacs executable.[1]

Optionally, Emacs can generate a *core dump* when it crashes. A core dump is a file containing voluminous data about the state of the program prior to the crash, usually examined by loading it into a debugger such as GDB. On many platforms, core dumps are disabled by default, and you must explicitly enable them by running the shell command '`ulimit -c unlimited`' (e.g., in your shell startup script).

### 34.2.7 Recovery After a Crash

If Emacs or the computer crashes, you can recover the files you were editing at the time of the crash from their auto-save files. To do this, start Emacs again and type the command `M-x recover-session`.

This command initially displays a buffer which lists interrupted session files, each with its date. You must choose which session to recover from. Typically the one you want is the most recent one. Move point to the one you choose, and type `C-c C-c`.

Then `recover-session` considers each of the files that you were editing during that session; for each such file, it asks whether to recover that file. If you answer `y` for a file, it shows the dates of that file and its auto-save file, then asks once again whether to recover that file. For the second question, you must confirm with `yes`. If you do, Emacs visits the file but gets the text from the auto-save file.

When `recover-session` is done, the files you've chosen to recover are present in Emacs buffers. You should then save them. Only this—saving them—updates the files themselves.

As a last resort, if you had buffers with content which were not associated with any files, or if the autosave was not recent enough to have recorded important changes, you can use the '`etc/emacs-buffer.gdb`' script with GDB (the GNU Debugger) to retrieve them from a core dump–provided that a core dump was saved, and that the Emacs executable was not stripped of its debugging symbols.

As soon as you get the core dump, rename it to another name such as '`core.emacs`', so that another crash won't overwrite it.

To use this script, run `gdb` with the file name of your Emacs executable and the file name of the core dump, e.g., '`gdb /usr/bin/emacs core.emacs`'. At the `(gdb)` prompt, load the recovery script: '`source /usr/src/emacs/etc/emacs-buffer.gdb`'. Then type the command `ybuffer-list` to see which buffers are available. For each buffer, it lists a buffer number. To save a buffer, use `ysave-buffer`; you specify the buffer number, and the file name to write that buffer into. You should use a file name which does not already exist; if the file does exist, the script does not make a backup of its old contents.

[1] You may wish to add the '`-p`' option, if your version of `addr2line` supports it.

### 34.2.8 Emergency Escape

On text terminals, the *emergency escape* feature suspends Emacs immediately if you type `C-g` a second time before Emacs can actually respond to the first one by quitting. This is so you can always get out of GNU Emacs no matter how badly it might be hung. When things are working properly, Emacs recognizes and handles the first `C-g` so fast that the second one won't trigger emergency escape. However, if some problem prevents Emacs from handling the first `C-g` properly, then the second one will get you back to the shell.

When you resume Emacs after a suspension caused by emergency escape, it asks two questions before going back to what it had been doing:

```
Auto-save? (y or n)
Abort (and dump core)? (y or n)
```

Answer each one with `y` or `n` followed by RET.

Saying `y` to '`Auto-save?`' causes immediate auto-saving of all modified buffers in which auto-saving is enabled. Saying `n` skips this.

Saying `y` to '`Abort (and dump core)?`' causes Emacs to crash, dumping core. This is to enable a wizard to figure out why Emacs was failing to quit in the first place. Execution does not continue after a core dump.

If you answer this question `n`, Emacs execution resumes. With luck, Emacs will ultimately do the requested quit. If not, each subsequent `C-g` invokes emergency escape again.

If Emacs is not really hung, just slow, you may invoke the double `C-g` feature without really meaning to. Then just resume and answer `n` to both questions, and you will get back to the former state. The quit you requested will happen by and by.

Emergency escape is active only for text terminals. On graphical displays, you can use the mouse to kill Emacs or switch to another program.

On MS-DOS, you must type `C-BREAK` (twice) to cause emergency escape—but there are cases where it won't work, when system call hangs or when Emacs is stuck in a tight loop in C code.

## 34.3 Reporting Bugs

If you think you have found a bug in Emacs, please report it. We cannot promise to fix it, or always to agree that it is a bug, but we certainly want to hear about it. The same applies for new features you would like to see added. The following sections will help you to construct an effective bug report.

### 34.3.1 Reading Existing Bug Reports and Known Problems

Before reporting a bug, if at all possible please check to see if it is already known about. Indeed, it may already have been fixed in a later release of Emacs, or in the development version. Here is a list of the main places you can read about known issues:

- The '`etc/PROBLEMS`' file; type `C-h C-p` to read it. This file contains a list of particularly well-known issues that have been encountered in compiling, installing and running Emacs. Often, there are suggestions for workarounds and solutions.
- Some additional user-level problems can be found in Section "Bugs and problems" in *GNU Emacs FAQ*.

- The GNU Bug Tracker at http://debbugs.gnu.org. Emacs bugs are filed in the tracker under the 'emacs' package. The tracker records information about the status of each bug, the initial bug report, and the follow-up messages by the bug reporter and Emacs developers. You can search for bugs by subject, severity, and other criteria.

  Instead of browsing the bug tracker as a webpage, you can browse it from Emacs using the debbugs package, which can be downloaded via the Package Menu (see ⟨undefined⟩ [Packages], page ⟨undefined⟩). This package provides the command M-x debbugs-gnu to list bugs, and M-x debbugs-gnu-search to search for a specific bug. User tags, applied by the Emacs maintainers, are shown by M-x debbugs-gnu-usertags.
- The 'bug-gnu-emacs' mailing list (also available as the newsgroup 'gnu.emacs.bug'). You can read the list archives at http://lists.gnu.org/mailman/listinfo/bug-gnu-emacs. This list works as a "mirror" of the Emacs bug reports and follow-up messages which are sent to the bug tracker. It also contains old bug reports from before the bug tracker was introduced (in early 2008).

  If you like, you can subscribe to the list. Be aware that its purpose is to provide the Emacs maintainers with information about bugs and feature requests, so reports may contain fairly large amounts of data; spectators should not complain about this.
- The 'emacs-pretest-bug' mailing list. This list is no longer used, and is mainly of historical interest. At one time, it was used for bug reports in development (i.e., not yet released) versions of Emacs. You can read the archives for 2003 to mid 2007 at http://lists.gnu.org/archive/html/emacs-pretest-bug/. Nowadays, it is an alias for 'bug-gnu-emacs'.
- The 'emacs-devel' mailing list. Sometimes people report bugs to this mailing list. This is not the main purpose of the list, however, and it is much better to send bug reports to the bug list. You should not feel obliged to read this list before reporting a bug.

### 34.3.2 When Is There a Bug

If Emacs accesses an invalid memory location ("segmentation fault"), or exits with an operating system error message that indicates a problem in the program (as opposed to something like "disk full"), then it is certainly a bug.

If the Emacs display does not correspond properly to the contents of the buffer, then it is a bug. But you should check that features like buffer narrowing (see ⟨undefined⟩ [Narrowing], page ⟨undefined⟩), which can hide parts of the buffer or change how it is displayed, are not responsible.

Taking forever to complete a command can be a bug, but you must make sure that it is really Emacs's fault. Some commands simply take a long time. Type C-g (C-BREAK on MS-DOS) and then C-h l to see whether the input Emacs received was what you intended to type; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an Emacs error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, it might actually

be working right. If in doubt, read the command's documentation (see ⟨undefined⟩ [Name Help], page ⟨undefined⟩).

A command's intended definition may not be the best possible definition for editing with. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. Ask other Emacs users, too. If you are not sure what the command is supposed to do after a careful reading of the manual, check the index and glossary for any terms that may be unclear.

If after careful rereading of the manual you still do not understand what the command should do, that indicates a bug in the manual, which you should report. The manual's job is to make everything clear to people who are not Emacs experts—including you. It is just as important to report documentation bugs as program bugs.

If the built-in documentation for a function or variable disagrees with the manual, one of them must be wrong; that is a bug.

### 34.3.3 Understanding Bug Reporting

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run Emacs, until the problem happens.

The most important principle in reporting a bug is to report *facts*. Hypotheses and verbal descriptions are no substitute for the detailed raw data. Reporting the facts is straightforward, but many people strain to posit explanations and report them instead of the facts. If the explanations are based on guesses about how Emacs is implemented, they will be useless; meanwhile, lacking the facts, we will have no real information about the bug. If you want to actually *debug* the problem, and report explanations that are more than guesses, that is useful—but please include the raw facts as well.

For example, suppose that you type `C-x C-f /glorp/baz.ugh RET`, visiting a file which (you know) happens to be rather large, and Emacs displays '`I feel pretty today`'. The bug report would need to provide all that information. You should not assume that the problem is due to the size of the file and say, "I visited a large file, and Emacs displayed '`I feel pretty today`'." This is what we mean by "guessing explanations". The problem might be due to the fact that there is a '`z`' in the file name. If this is so, then when we got your report, we would try out the problem with some "large file", probably with no '`z`' in its name, and not see any problem. There is no way we could guess that we should try visiting a file with a '`z`' in its name.

You should not even say "visit a file" instead of `C-x C-f`. Similarly, rather than saying "if I have three characters on the line", say "after I type `RET A B C RET C-p`", if that is the way you entered the text.

If possible, try quickly to reproduce the bug by invoking Emacs with `emacs -Q` (so that Emacs starts with no initial customizations; see Section E.2 [Initial Options], page 755), and repeating the steps that you took to trigger the bug. If you can reproduce the bug this way, that rules out bugs in your personal customizations. Then your bug report should begin by stating that you started Emacs with `emacs -Q`, followed by the exact sequence of

steps for reproducing the bug. If possible, inform us of the exact contents of any file that is needed to reproduce the bug.

Some bugs are not reproducible from `emacs -Q`; some are not easily reproducible at all. In that case, you should report what you have—but, as before, please stick to the raw facts about what you did to trigger the bug the first time.

### 34.3.4 Checklist for Bug Reports

Before reporting a bug, first try to see if the problem has already been reported (see Section 34.3.1 [Known Problems], page 722).

If you are able to, try the latest release of Emacs to see if the problem has already been fixed. Even better is to try the latest development version. We recognize that this is not easy for some people, so do not feel that you absolutely must do this before making a report.

The best way to write a bug report for Emacs is to use the command `M-x report-emacs-bug`. This sets up a mail buffer (see Chapter 29 [Sending Mail], page 624) and automatically inserts *some* of the essential information. However, it cannot supply all the necessary information; you should still read and follow the guidelines below, so you can enter the other crucial information by hand before you send the message. You may feel that some of the information inserted by `M-x report-emacs-bug` is not relevant, but unless you are absolutely sure it is best to leave it, so that the developers can decide for themselves.

When you have finished writing your report, type `C-c C-c` and it will be sent to the Emacs maintainers at `bug-gnu-emacs@gnu.org`. (If you want to suggest an improvement or new feature, use the same address.) If you cannot send mail from inside Emacs, you can copy the text of your report to your normal mail client (if your system supports it, you can type `C-c M-i` to have Emacs do this for you) and send it to that address. Or you can simply send an email to that address describing the problem.

Your report will be sent to the '`bug-gnu-emacs`' mailing list, and stored in the GNU Bug Tracker at `http://debbugs.gnu.org`. Please include a valid reply email address, in case we need to ask you for more information about your report. Submissions are moderated, so there may be a delay before your report appears.

You do not need to know how the Gnu Bug Tracker works in order to report a bug, but if you want to, you can read the tracker's online documentation to see the various features you can use.

All mail sent to the '`bug-gnu-emacs`' mailing list is also gatewayed to the '`gnu.emacs.bug`' newsgroup. The reverse is also true, but we ask you not to post bug reports (or replies) via the newsgroup. It can make it much harder to contact you if we need to ask for more information, and it does not integrate well with the bug tracker.

If your data is more than 500,000 bytes, please don't include it directly in the bug report; instead, offer to send it on request, or make it available by ftp and say where.

To enable maintainers to investigate a bug, your report should include all these things:

- The version number of Emacs. Without this, we won't know whether there is any point in looking for the bug in the current version of GNU Emacs.

  `M-x report-emacs-bug` includes this information automatically, but if you are not using that command for your report you can get the version number by typing `M-x`

`emacs-version RET`. If that command does not work, you probably have something other than GNU Emacs, so you will have to report the bug somewhere else.

- The type of machine you are using, and the operating system name and version number (again, automatically included by `M-x report-emacs-bug`). `M-x emacs-version RET` provides this information too. Copy its output from the '`*Messages*`' buffer, so that you get it all and get it accurately.
- The operands given to the `configure` command when Emacs was installed (automatically included by `M-x report-emacs-bug`).
- A complete list of any modifications you have made to the Emacs source. (We may not have time to investigate the bug unless it happens in an unmodified Emacs. But if you've made modifications and you don't tell us, you are sending us on a wild goose chase.)

  Be precise about these changes. A description in English is not enough—send a context diff for them.

  Adding files of your own, or porting to another machine, is a modification of the source.
- Details of any other deviations from the standard procedure for installing GNU Emacs.
- The complete text of any files needed to reproduce the bug.

  If you can tell us a way to cause the problem without visiting any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can matter whether there are spaces at the ends of lines, or a newline after the last line in the buffer (nothing ought to care whether the last line is terminated, but try telling the bugs that).
- The precise commands we need to type to reproduce the bug. If at all possible, give a full recipe for an Emacs started with the '`-Q`' option (see Section E.2 [Initial Options], page 755). This bypasses your personal customizations.

  One way to record the input to Emacs precisely is to write a dribble file. To start the file, use the `M-x open-dribble-file RET` command. From then on, Emacs copies all your input to the specified dribble file until the Emacs process is killed.
- For possible display bugs, the terminal type (the value of environment variable `TERM`), the complete termcap entry for the terminal from '`/etc/termcap`' (since that file is not identical on all machines), and the output that Emacs actually sent to the terminal.

  The way to collect the terminal output is to execute the Lisp expression

  ```
  (open-termscript "~/termscript")
  ```

  using `M-:` or from the '`*scratch*`' buffer just after starting Emacs. From then on, Emacs copies all terminal output to the specified termscript file as well, until the Emacs process is killed. If the problem happens when Emacs starts up, put this expression into your Emacs initialization file so that the termscript file will be open when Emacs displays the screen for the first time.

  Be warned: it is often difficult, and sometimes impossible, to fix a terminal-dependent bug without access to a terminal of the type that stimulates the bug.
- If non-ASCII text or internationalization is relevant, the locale that was current when you started Emacs. On GNU/Linux and Unix systems, or if you use a Posix-style shell such as Bash, you can use this shell command to view the relevant values:

```
echo LC_ALL=$LC_ALL LC_COLLATE=$LC_COLLATE LC_CTYPE=$LC_CTYPE \
  LC_MESSAGES=$LC_MESSAGES LC_TIME=$LC_TIME LANG=$LANG
```

Alternatively, use the `locale` command, if your system has it, to display your locale settings.

You can use the `M-!` command to execute these commands from Emacs, and then copy the output from the '`*Messages*`' buffer into the bug report. Alternatively, `M-x getenv RET LC_ALL RET` will display the value of `LC_ALL` in the echo area, and you can copy its output from the '`*Messages*`' buffer.

- A description of what behavior you observe that you believe is incorrect. For example, "The Emacs process gets a fatal signal", or, "The resulting text is as follows, which I think is wrong."

  Of course, if the bug is that Emacs gets a fatal signal, then one can't miss it. But if the bug is incorrect text, the maintainer might fail to notice what is wrong. Why leave it to chance?

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the source is out of sync, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and the copy here might not. If you *said* to expect a crash, then when Emacs here fails to crash, we would know that the bug was not happening. If you don't say to expect a crash, then we would not know whether the bug was happening—we would not be able to draw any conclusion from our observations.
- If the bug is that the Emacs Manual or the Emacs Lisp Reference Manual fails to describe the actual behavior of Emacs, or that the text is confusing, copy in the text from the online manual which you think is at fault. If the section is small, just the section name is enough.
- If the manifestation of the bug is an Emacs error message, it is important to report the precise text of the error message, and a backtrace showing how the Lisp program in Emacs arrived at the error.

  To get the error message text accurately, copy it from the '`*Messages*`' buffer into the bug report. Copy all of it, not just part.

  To make a backtrace for the error, use `M-x toggle-debug-on-error` before the error happens (that is to say, you must give that command and then make the bug happen). This causes the error to start the Lisp debugger, which shows you a backtrace. Copy the text of the debugger's backtrace into the bug report. See Section "The Lisp Debugger" in *the Emacs Lisp Reference Manual*, for information on debugging Emacs Lisp programs with the Edebug package.

  This use of the debugger is possible only if you know how to make the bug happen again. If you can't make it happen again, at least copy the whole error message.

  If Emacs appears to be stuck in an infinite loop or in a very long operation, typing `C-g` with the variable `debug-on-quit` non-`nil` will start the Lisp debugger and show a backtrace. This backtrace is useful for debugging such long loops, so if you can produce it, copy it into the bug report.

  If you cannot get Emacs to respond to `C-g` (e.g., because `inhibit-quit` is set), then you can try sending the signal specified by `debug-on-event` (default SIGUSR2) from outside Emacs to cause it to enter the debugger.

- Check whether any programs you have loaded into the Lisp world, including your initialization file, set any variables that may affect the functioning of Emacs. Also, see whether the problem happens in a freshly started Emacs without loading your initialization file (start Emacs with the `-Q` switch to prevent loading the init files). If the problem does *not* occur then, you must report the precise contents of any programs that you must load into the Lisp world in order to cause the problem to occur.
- If the problem does depend on an init file or other Lisp programs that are not part of the standard Emacs system, then you should make sure it is not a bug in those programs by complaining to their maintainers first. After they verify that they are using Emacs in a way that is supposed to work, they should report the bug.
- If you wish to mention something in the GNU Emacs source, show the line of code with a few lines of context. Don't just give a line number.

  The line numbers in the development sources don't match those in your sources. It would take extra work for the maintainers to determine what code is in your version at a given line number, and we could not be certain.
- Additional information from a C debugger such as GDB might enable someone to find a problem on a machine which he does not have available. If you don't know how to use GDB, please read the GDB manual—it is not very long, and using GDB is easy. You can find the GDB distribution, including the GDB manual in online form, in most of the same places you can find the Emacs distribution. To run Emacs under GDB, you should switch to the '`src`' subdirectory in which Emacs was compiled, then do '`gdb emacs`'. It is important for the directory '`src`' to be current so that GDB will read the '`.gdbinit`' file in this directory.

  However, you need to think when you collect the additional information if you want it to show what causes the bug.

  For example, many people send just a backtrace, but that is not very useful by itself. A simple backtrace with arguments often conveys little about what is happening inside GNU Emacs, because most of the arguments listed in the backtrace are pointers to Lisp objects. The numeric values of these pointers have no significance whatever; all that matters is the contents of the objects they point to (and most of the contents are themselves pointers).

  To provide useful information, you need to show the values of Lisp objects in Lisp notation. Do this for each variable which is a Lisp object, in several stack frames near the bottom of the stack. Look at the source to see which variables are Lisp objects, because the debugger thinks of them as integers.

  To show a variable's value in Lisp syntax, first print its value, then use the user-defined GDB command `pr` to print the Lisp object in Lisp syntax. (If you must use another debugger, call the function `debug_print` with the object as an argument.) The `pr` command is defined by the file '`.gdbinit`', and it works only if you are debugging a running process (not with a core dump).

  To make Lisp errors stop Emacs and return to GDB, put a breakpoint at `Fsignal`.

  For a short listing of Lisp functions running, type the GDB command `xbacktrace`.

  The file '`.gdbinit`' defines several other commands that are useful for examining the data types and contents of Lisp objects. Their names begin with '`x`'. These commands

work at a lower level than `pr`, and are less convenient, but they may work even when `pr` does not, such as when debugging a core dump or when Emacs has had a fatal signal.

More detailed advice and other useful techniques for debugging Emacs are available in the file '`etc/DEBUG`' in the Emacs distribution. That file also includes instructions for investigating problems whereby Emacs stops responding (many people assume that Emacs is "hung", whereas in fact it might be in an infinite loop).

To find the file '`etc/DEBUG`' in your Emacs installation, use the directory name stored in the variable `data-directory`.

Here are some things that are not necessary in a bug report:

- A description of the envelope of the bug—this is not necessary for a reproducible bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time-consuming and not very useful, because the way we will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples. You might as well save time by not searching for additional examples. It is better to send the bug report right away, go back to editing, and find another bug to report.

  Of course, if you can find a simpler example to report *instead* of the original one, that is a convenience. Errors in the output will be easier to spot, running under the debugger will take less time, etc.

  However, simplification is not vital; if you can't do this or don't have time to try, please report the bug with your original test case.

- A core dump file.

  Debugging the core dump might be useful, but it can only be done on your machine, with your Emacs executable. Therefore, sending the core dump file to the Emacs maintainers won't be useful. Above all, don't include the core file in an email bug report! Such a large message can be extremely inconvenient.

- A system-call trace of Emacs execution.

  System-call traces are very useful for certain special kinds of debugging, but in most cases they give little useful information. It is therefore strange that many people seem to think that *the* way to report information about a crash is to send a system-call trace. Perhaps this is a habit formed from experience debugging programs that don't have source code or debugging symbols.

  In most programs, a backtrace is normally far, far more informative than a system-call trace. Even in Emacs, a simple backtrace is generally more informative, though to give full information you should supplement the backtrace by displaying variable values and printing them as Lisp objects with `pr` (see above).

- A patch for the bug.

  A patch for the bug is useful if it is a good one. But don't omit the other information that a bug report needs, such as the test case, on the assumption that a patch is sufficient. We might see problems with your patch and decide to fix the problem another way, or we might not understand it at all. And if we can't understand what bug you are trying to fix, or why your patch should be an improvement, we mustn't install it.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even experts can't guess right about such things without first using the debugger to find the facts.

### 34.3.5 Sending Patches for GNU Emacs

If you would like to write bug fixes or improvements for GNU Emacs, that is very helpful. When you send your changes, please follow these guidelines to make it easy for the maintainers to use them. If you don't follow these guidelines, your information might still be useful, but using it will take extra work. Maintaining GNU Emacs is a lot of work in the best of circumstances, and we can't keep up unless you do your best to help.

- Send an explanation with your changes of what problem they fix or what improvement they bring about. For a fix for an existing bug, it is best to reply to the relevant discussion on the '`bug-gnu-emacs`' list, or the bug entry in the GNU Bug Tracker at `http://debbugs.gnu.org`. Explain why your change fixes the bug.
- Always include a proper bug report for the problem you think you have fixed. We need to convince ourselves that the change is right before installing it. Even if it is correct, we might have trouble understanding it if we don't have a way to reproduce the problem.
- Include all the comments that are appropriate to help people reading the source in the future understand why this change was needed.
- Don't mix together changes made for different reasons. Send them *individually*.

  If you make two changes for separate reasons, then we might not want to install them both. We might want to install just one. If you send them all jumbled together in a single set of diffs, we have to do extra work to disentangle them—to figure out which parts of the change serve which purpose. If we don't have time for this, we might have to ignore your changes entirely.

  If you send each change as soon as you have written it, with its own explanation, then two changes never get tangled up, and we can consider each one properly without any extra work to disentangle them.
- Send each change as soon as that change is finished. Sometimes people think they are helping us by accumulating many changes to send them all together. As explained above, this is absolutely the worst thing you could do.

  Since you should send each change separately, you might as well send it right away. That gives us the option of installing it immediately if it is important.
- Use '`diff -c`' to make your diffs. Diffs without context are hard to install reliably. More than that, they are hard to study; we must always study a patch to decide whether we want to install it. Unidiff format is better than contextless diffs, but not as easy to read as '`-c`' format.

  If you have GNU diff, use '`diff -c -F'^[_a-zA-Z0-9$]+ *('`' when making diffs of C code. This shows the name of the function that each change occurs in.
- Avoid any ambiguity as to which is the old version and which is the new. Please make the old version the first argument to diff, and the new version the second argument. And please give one version or the other a name that indicates whether it is the old version or your new changed one.

- Write the change log entries for your changes. This is both to save us the extra work of writing them, and to help explain your changes so we can understand them.

  The purpose of the change log is to show people where to find what was changed. So you need to be specific about what functions you changed; in large functions, it's often helpful to indicate where within the function the change was.

  On the other hand, once you have shown people where to find the change, you need not explain its purpose in the change log. Thus, if you add a new function, all you need to say about it is that it is new. If you feel that the purpose needs explaining, it probably does—but put the explanation in comments in the code. It will be more useful there.

  Please read the '`ChangeLog`' files in the '`src`' and '`lisp`' directories to see what sorts of information to put in, and to learn the style that we use. See Section 25.2 [Change Log], page 570.
- When you write the fix, keep in mind that we can't install a change that would break other systems. Please think about what effect your change will have if compiled on another type of system.

  Sometimes people send fixes that *might* be an improvement in general—but it is hard to be sure of this. It's hard to install such changes because we have to study them very carefully. Of course, a good explanation of the reasoning by which you concluded the change was correct can help convince us.

  The safest changes are changes to the configuration files for a particular machine. These are safe because they can't create new bugs on other machines.

  Please help us keep up with the workload by designing the patch in a form that is clearly safe to install.

## 34.4 Contributing to Emacs Development

If you would like to help pretest Emacs releases to assure they work well, or if you would like to work on improving Emacs, please contact the maintainers at `emacs-devel@gnu.org`. A pretester should be prepared to investigate bugs as well as report them. If you'd like to work on improving Emacs, please ask for suggested projects or suggest your own ideas.

If you have already written an improvement, please tell us about it. If you have not yet started work, it is useful to contact `emacs-devel@gnu.org` before you start; it might be possible to suggest ways to make your extension fit in better with the rest of Emacs.

The development version of Emacs can be downloaded from the repository where it is actively maintained by a group of developers. See the Emacs project page `http://savannah.gnu.org/projects/emacs/` for details.

For more information on how to contribute, see the '`etc/CONTRIBUTE`' file in the Emacs distribution.

## 34.5 How To Get Help with GNU Emacs

If you need help installing, using or changing GNU Emacs, there are two ways to find it:

- Send a message to the mailing list `help-gnu-emacs@gnu.org`, or post your request on newsgroup `gnu.emacs.help`. (This mailing list and newsgroup interconnect, so it does not matter which one you use.)

- Look in the service directory for someone who might help you for a fee. The service directory is found in the file named ‘`etc/SERVICE`’ in the Emacs distribution.

# Appendix A GNU GENERAL PUBLIC LICENSE

# Appendix B GNU General Public License

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

## Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

## TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.
   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

## END OF TERMS AND CONDITIONS

### How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.

# Appendix C GNU Free Documentation License

# Appendix D GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008, 2009 Free Software Foundation, Inc.
http://fsf.org/

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

## ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

# Appendix E Command Line Arguments for Emacs Invocation

Emacs supports command line arguments to request various actions when invoking Emacs. These are for compatibility with other editors and for sophisticated activities. We don't recommend using them for ordinary editing (See Section 31.4 [Emacs Server], page 665, for a way to access an existing Emacs job from the command line).

Arguments starting with '`-`' are *options*, and so is '`+linenum`'. All other arguments specify files to visit. Emacs visits the specified files while it starts up. The last file specified on the command line becomes the current buffer; the other files are also visited in other buffers. As with most programs, the special argument '`--`' says that all subsequent arguments are file names, not options, even if they start with '`-`'.

Emacs command options can specify many things, such as the size and position of the X window Emacs uses, its colors, and so on. A few options support advanced usage, such as running Lisp functions on files in batch mode. The sections of this chapter describe the available options, arranged according to their purpose.

There are two ways of writing options: the short forms that start with a single '`-`', and the long forms that start with '`--`'. For example, '`-d`' is a short form and '`--display`' is the corresponding long form.

The long forms with '`--`' are easier to remember, but longer to type. However, you don't have to spell out the whole option name; any unambiguous abbreviation is enough. When a long option takes an argument, you can use either a space or an equal sign to separate the option name and the argument. Thus, you can write either '`--display sugar-bombs:0.0`' or '`--display=sugar-bombs:0.0`'. We recommend an equal sign because it makes the relationship clearer, and the tables below always show an equal sign.

Most options specify how to initialize Emacs, or set parameters for the Emacs session. We call them *initial options*. A few options specify things to do, such as loading libraries or calling Lisp functions. These are called *action options*. These and file names together are called *action arguments*. The action arguments are stored as a list of strings in the variable `command-line-args`. (Actually, when Emacs starts up, `command-line-args` contains all the arguments passed from the command line; during initialization, the initial arguments are removed from this list when they are processed, leaving only the action arguments.)

## E.1 Action Arguments

Here is a table of action arguments:

'*file*'
'`--file=`*file*'
'`--find-file=`*file*'
'`--visit=`*file*'

Visit *file* using `find-file`. See ⟨undefined⟩ [Visiting], page ⟨undefined⟩.

When Emacs starts up, it displays the startup buffer in one window, and the buffer visiting *file* in another window (see Chapter 17 [Windows], page 289). If you supply more than one file argument, the displayed file is the last one specified on the command line; the other files are visited but their buffers are not shown.

If the startup buffer is disabled (see Section 3.1 [Entering Emacs], page 56), then *file* is visited in a single window if one file argument was supplied; with two file arguments, Emacs displays the files in two different windows; with more than two file argument, Emacs displays the last file specified in one window, plus a Buffer Menu in a different window (see ⟨undefined⟩ [Several Buffers], page ⟨undefined⟩). To inhibit using the Buffer Menu for this, change the variable `inhibit-startup-buffer-menu` to `t`.

'`+`*linenum file*'
: Visit *file* using `find-file`, then go to line number *linenum* in it.

'`+`*linenum*`:`*columnnum file*'
: Visit *file* using `find-file`, then go to line number *linenum* and put point at column number *columnnum*.

'`-l` *file*'
'`--load=`*file*'
: Load a Lisp library named *file* with the function `load`. If *file* is not an absolute file name, Emacs first looks for it in the current directory, then in the directories listed in `load-path` (see Section 24.8 [Lisp Libraries], page 549).

  **Warning:** If previous command-line arguments have visited files, the current directory is the directory of the last file visited.

'`-L` *dir*'
'`--directory=`*dir*'
: Add directory *dir* to the variable `load-path`.

'`-f` *function*'
'`--funcall=`*function*'
: Call Lisp function *function*. If it is an interactive function (a command), it reads the arguments interactively just as if you had called the same function with a key sequence. Otherwise, it calls the function with no arguments.

'`--eval=`*expression*'
'`--execute=`*expression*'
: Evaluate Lisp expression *expression*.

'`--insert=`*file*'
: Insert the contents of *file* into the '`*scratch*`' buffer (see Section 24.10 [Lisp Interaction], page 552). This is like what `M-x insert-file` does (see ⟨undefined⟩ [Misc File Ops], page ⟨undefined⟩).

'`--kill`'
: Exit from Emacs without asking for confirmation.

'`--help`'
: Print a usage message listing all available options, then exit successfully.

'`--version`'
: Print Emacs version, then exit successfully.

## E.2 Initial Options

The initial options specify parameters for the Emacs session. This section describes the more general initial options; some other options specifically related to the X Window System appear in the following sections.

Some initial options affect the loading of the initialization file. Normally, Emacs first loads '`site-start.el`' if it exists, then your own initialization file if it exists, and finally the default initialization file '`default.el`' if it exists (see Section 33.4 [Init File], page 711). Certain options prevent loading of some of these files or substitute other files for them.

'`-chdir` *directory*'
'`--chdir=`*directory*'
: Change to *directory* before doing anything else. This is mainly used by session management in X so that Emacs starts in the same directory as it stopped. This makes desktop saving and restoring easier.

'`-t` *device*'
'`--terminal=`*device*'
: Use *device* as the device for terminal input and output. This option implies '`--no-window-system`'.

'`-d` *display*'
'`--display=`*display*'
: Use the X Window System and use the display named *display* to open the initial Emacs frame. See Section E.5 [Display X], page 762, for more details.

'`-nw`'
'`--no-window-system`'
: Don't communicate directly with the window system, disregarding the `DISPLAY` environment variable even if it is set. This means that Emacs uses the terminal from which it was launched for all its display and input.

'`-batch`'
'`--batch`'
: Run Emacs in *batch mode*. Batch mode is used for running programs written in Emacs Lisp from shell scripts, makefiles, and so on. To invoke a Lisp program, use the '`-batch`' option in conjunction with one or more of '`-l`', '`-f`' or '`--eval`' (see Section E.1 [Action Arguments], page 754). See Section E.3 [Command Example], page 758, for an example.

  In batch mode, Emacs does not display the text being edited, and the standard terminal interrupt characters such as `C-z` and `C-c` have their usual effect. Emacs functions that normally print a message in the echo area will print to either the standard output stream (`stdout`) or the standard error stream (`stderr`) instead. (To be precise, functions like `prin1`, `princ` and `print` print to `stdout`, while `message` and `error` print to `stderr`.) Functions that normally read keyboard input from the minibuffer take their input from the terminal's standard input stream (`stdin`) instead.

  '`--batch`' implies '`-q`' (do not load an initialization file), but '`site-start.el`' is loaded nonetheless. It also causes Emacs to exit after processing all the command options. In addition, it disables auto-saving except in buffers for which auto-saving is explicitly requested.

'`--script` *file*'
: Run Emacs in batch mode, like '`--batch`', and then read and execute the Lisp code in *file*.

The normal use of this option is in executable script files that run Emacs. They can start with this text on the first line

```
#!/usr/bin/emacs --script
```

which will invoke Emacs with '`--script`' and supply the name of the script file as *file*. Emacs Lisp then treats the '`#!`' on this first line as a comment delimiter.

'`-q`'
'`--no-init-file`'
: Do not load any initialization file (see Section 33.4 [Init File], page 711). When Emacs is invoked with this option, the Customize facility does not allow options to be saved (see Section 33.1 [Easy Customization], page 686). This option does not disable loading '`site-start.el`'.

'`--no-site-file`'
: Do not load '`site-start.el`' (see Section 33.4 [Init File], page 711). The '`-Q`' option does this too, but other options like '`-q`' do not.

'`--no-site-lisp`'
: Do not include the '`site-lisp`' directories in `load-path` (see Section 33.4 [Init File], page 711). The '`-Q`' option does this too.

'`--no-splash`'
: Do not display a startup screen. You can also achieve this effect by setting the variable `inhibit-startup-screen` to non-`nil` in your initialization file (see Section 3.1 [Entering Emacs], page 56).

'`-Q`'
'`--quick`'
: Start emacs with minimum customizations. This is similar to using '`-q`', '`--no-site-file`', '`--no-site-lisp`', and '`--no-splash`' together. This also stops Emacs from processing X resources by setting `inhibit-x-resources` to `t` (see Section F.1 [Resources], page 769).

'`-daemon`'
'`--daemon`'
: Start Emacs as a daemon—after Emacs starts up, it starts the Emacs server and disconnects from the terminal without opening any frames. You can then use the `emacsclient` command to connect to Emacs for editing. See Section 31.4 [Emacs Server], page 665, for information about using Emacs as a daemon.

'`-daemon=`*`SERVER-NAME`*'
: Start emacs in background as a daemon, and use *SERVER-NAME* as the server name.

'`--no-desktop`'
: Do not reload any saved desktop. See Section 31.8 [Saving Emacs Sessions], page 675.

'`-u` *`user`*'
'`--user=`*`user`*'
: Load *user*'s initialization file instead of your own[1].

[1] This option has no effect on MS-Windows.

‘`--debug-init`’
: Enable the Emacs Lisp debugger for errors in the init file. See Section “Entering the Debugger on an Error” in *The GNU Emacs Lisp Reference Manual*.

## E.3 Command Argument Example

Here is an example of using Emacs with arguments and options. It assumes you have a Lisp program file called ‘`hack-c.el`’ which, when loaded, performs some useful operation on the current buffer, expected to be a C program.

```
emacs --batch foo.c -l hack-c -f save-buffer >& log
```

This says to visit ‘`foo.c`’, load ‘`hack-c.el`’ (which makes changes in the visited file), save ‘`foo.c`’ (note that `save-buffer` is the function that `C-x C-s` is bound to), and then exit back to the shell (because of ‘`--batch`’). ‘`--batch`’ also guarantees there will be no problem redirecting output to ‘`log`’, because Emacs will not assume that it has a display terminal to work with.

## E.4 Environment Variables

The *environment* is a feature of the operating system; it consists of a collection of variables with names and values. Each variable is called an *environment variable*; environment variable names are case-sensitive, and it is conventional to use upper case letters only. The values are all text strings.

What makes the environment useful is that subprocesses inherit the environment automatically from their parent process. This means you can set up an environment variable in your login shell, and all the programs you run (including Emacs) will automatically see it. Subprocesses of Emacs (such as shells, compilers, and version control programs) inherit the environment from Emacs, too.

Inside Emacs, the command `M-x getenv` reads the name of an environment variable, and prints its value in the echo area. `M-x setenv` sets a variable in the Emacs environment, and `C-u M-x setenv` removes a variable. (Environment variable substitutions with ‘`$`’ work in the value just as in file names; see ⟨undefined⟩ [File Names with $], page ⟨undefined⟩.) The variable `initial-environment` stores the initial environment inherited by Emacs.

The way to set environment variables outside of Emacs depends on the operating system, and especially the shell that you are using. For example, here’s how to set the environment variable `ORGANIZATION` to ‘`not very much`’ using Bash:

```
export ORGANIZATION="not very much"
```

and here’s how to do it in csh or tcsh:

```
setenv ORGANIZATION "not very much"
```

When Emacs is using the X Window System, various environment variables that control X work for Emacs as well. See the X documentation for more information.

### E.4.1 General Variables

Here is an alphabetical list of environment variables that have special meanings in Emacs. Most of these variables are also used by some other programs. Emacs does not require any of these environment variables to be set, but it uses their values if they are set.

`CDPATH` Used by the `cd` command to search for the directory you specify, when you specify a relative directory name.

`DBUS_SESSION_BUS_ADDRESS`
Used by D-Bus when Emacs is compiled with it. Usually, there is no need to change it. Setting it to a dummy address, like '`unix:path=/tmp/foo`', suppresses connections to the D-Bus session bus.

`EMACSDATA`
Directory for the architecture-independent files that come with Emacs. This is used to initialize the variable `data-directory`.

`EMACSDOC` Directory for the documentation string file, which is used to initialize the Lisp variable `doc-directory`.

`EMACSLOADPATH`
A colon-separated list of directories[2] to search for Emacs Lisp files. If set, it overrides the usual initial value of the `load-path` variable (see Section 24.8 [Lisp Libraries], page 549).

`EMACSPATH`
A colon-separated list of directories to search for executable files. If set, Emacs uses this in addition to `PATH` (see below) when initializing the variable `exec-path` (see Section 31.3 [Shell], page 655).

`EMAIL` Your email address; used to initialize the Lisp variable `user-mail-address`, which the Emacs mail interface puts into the '`From`' header of outgoing messages (see Section 29.2 [Mail Headers], page 625).

`ESHELL` Used for shell-mode to override the `SHELL` environment variable (see Section 31.3.2 [Interactive Shell], page 656).

`HISTFILE` The name of the file that shell commands are saved in between logins. This variable defaults to '`~/.bash_history`' if you use Bash, to '`~/.sh_history`' if you use ksh, and to '`~/.history`' otherwise.

`HOME` The location of your files in the directory tree; used for expansion of file names starting with a tilde ('`~`'). On MS-DOS, it defaults to the directory from which Emacs was started, with '`/bin`' removed from the end if it was present. On Windows, the default value of `HOME` is the '`Application Data`' subdirectory of the user profile directory (normally, this is '`C:/Documents and Settings/`*`username`*`/Application Data`', where *username* is your user name), though for backwards compatibility '`C:/`' will be used instead if a '`.emacs`' file is found there.

`HOSTNAME` The name of the machine that Emacs is running on.

`INCPATH` A colon-separated list of directories. Used by the `complete` package to search for files.

[2] Here and below, whenever we say "colon-separated list of directories", it pertains to Unix and GNU/Linux systems. On MS-DOS and MS-Windows, the directories are separated by semi-colons instead, since DOS/Windows file names might include a colon after a drive letter.

`INFOPATH` A colon-separated list of directories in which to search for Info files.

`LC_ALL`
`LC_COLLATE`
`LC_CTYPE`
`LC_MESSAGES`
`LC_MONETARY`
`LC_NUMERIC`
`LC_TIME`
`LANG` The user's preferred locale. The locale has six categories, specified by the environment variables `LC_COLLATE` for sorting, `LC_CTYPE` for character encoding, `LC_MESSAGES` for system messages, `LC_MONETARY` for monetary formats, `LC_NUMERIC` for numbers, and `LC_TIME` for dates and times. If one of these variables is not set, the category defaults to the value of the `LANG` environment variable, or to the default '`C`' locale if `LANG` is not set. But if `LC_ALL` is specified, it overrides the settings of all the other locale environment variables.

On MS-Windows, if `LANG` is not already set in the environment when Emacs starts, Emacs sets it based on the system-wide default language, which you can set in the '`Regional Settings`' Control Panel on some versions of MS-Windows.

The value of the `LC_CTYPE` category is matched against entries in `locale-language-names`, `locale-charset-language-names`, and `locale-preferred-coding-systems`, to select a default language environment and coding system. See Section 19.3 [Language Environments], page 377.

`LOGNAME` The user's login name. See also `USER`.

`MAIL` The name of your system mail inbox.

`NAME` Your real-world name. This is used to initialize the variable `user-full-name` (see Section 29.2 [Mail Headers], page 625).

`NNTPSERVER`
The name of the news server. Used by the mh and Gnus packages.

`ORGANIZATION`
The name of the organization to which you belong. Used for setting the 'Organization:' header in your posts from the Gnus package.

`PATH` A colon-separated list of directories containing executable files. This is used to initialize the variable `exec-path` (see Section 31.3 [Shell], page 655).

`PWD` If set, this should be the default directory when Emacs was started.

`REPLYTO` If set, this specifies an initial value for the variable `mail-default-reply-to` (see Section 29.2 [Mail Headers], page 625).

`SAVEDIR` The name of a directory in which news articles are saved by default. Used by the Gnus package.

`SHELL` The name of an interpreter used to parse and execute programs run from inside Emacs.

`SMTPSERVER`
: The name of the outgoing mail server. This is used to initialize the variable `smtpmail-smtp-server` (see Section 29.4.1 [Mail Sending], page 627).

`TERM`
: The type of the terminal that Emacs is using. This variable must be set unless Emacs is run in batch mode. On MS-DOS, it defaults to '`internal`', which specifies a built-in terminal emulation that handles the machine's own display.

`TERMCAP`
: The name of the termcap library file describing how to program the terminal specified by `TERM`. This defaults to '`/etc/termcap`'.

`TMPDIR`
`TMP`
`TEMP`
: These environment variables are used to initialize the variable `temporary-file-directory`, which specifies a directory in which to put temporary files (see ⟨undefined⟩ [Backup], page ⟨undefined⟩). Emacs tries to use `TMPDIR` first; if that is unset, it tries `TMP`, then `TEMP`, and finally '`/tmp`'. But on MS-Windows and MS-DOS, Emacs tries `TEMP`, then `TMPDIR`, then `TMP`, and finally '`c:/temp`'.

`TZ`
: This specifies the current time zone and possibly also daylight saving time information. On MS-DOS, if `TZ` is not set in the environment when Emacs starts, Emacs defines a default value as appropriate for the country code returned by DOS. On MS-Windows, Emacs does not use `TZ` at all.

`USER`
: The user's login name. See also `LOGNAME`. On MS-DOS, this defaults to '`root`'.

`VERSION_CONTROL`
: Used to initialize the `version-control` variable (see ⟨undefined⟩ [Backup Names], page ⟨undefined⟩).

### E.4.2 Miscellaneous Variables

These variables are used only on particular configurations:

`COMSPEC`
: On MS-DOS and MS-Windows, the name of the command interpreter to use when invoking batch files and commands internal to the shell. On MS-DOS this is also used to make a default value for the `SHELL` environment variable.

`NAME`
: On MS-DOS, this variable defaults to the value of the `USER` variable.

`EMACSTEST`
: On MS-DOS, this specifies a file to use to log the operation of the internal terminal emulator. This feature is useful for submitting bug reports.

`EMACSCOLORS`
: On MS-DOS, this specifies the screen colors. It is useful to set them this way, since otherwise Emacs would display the default colors momentarily when it starts up.

  The value of this variable should be the two-character encoding of the foreground (the first character) and the background (the second character) colors of the default face. Each character should be the hexadecimal code for the desired color on a standard PC text-mode display. For example, to get blue text on a light gray background, specify '`EMACSCOLORS=17`', since 1 is the code of the blue color and 7 is the code of the light gray color.

The PC display usually supports only eight background colors. However, Emacs switches the DOS display to a mode where all 16 colors can be used for the background, so all four bits of the background color are actually used.

`PRELOAD_WINSOCK`
: On MS-Windows, if you set this variable, Emacs will load and initialize the network library at startup, instead of waiting until the first time it is required.

`emacs_dir`
: On MS-Windows, `emacs_dir` is a special environment variable, which indicates the full path of the directory in which Emacs is installed. If Emacs is installed in the standard directory structure, it calculates this value automatically. It is not much use setting this variable yourself unless your installation is non-standard, since unlike other environment variables, it will be overridden by Emacs at startup. When setting other environment variables, such as `EMACSLOADPATH`, you may find it useful to use `emacs_dir` rather than hard-coding an absolute path. This allows multiple versions of Emacs to share the same environment variable settings, and it allows you to move the Emacs installation directory, without changing any environment or registry settings.

### E.4.3 The MS-Windows System Registry

On MS-Windows, the installation program `addpm.exe` adds values for `emacs_dir`, `EMACSLOADPATH`, `EMACSDATA`, `EMACSPATH`, `EMACSDOC`, `SHELL` and `TERM` to the '`HKEY_LOCAL_MACHINE`' section of the system registry, under '`/Software/GNU/Emacs`'. It does this because there is no standard place to set environment variables across different versions of Windows. Running `addpm.exe` is no longer strictly necessary in recent versions of Emacs, but if you are upgrading from an older version, running `addpm.exe` ensures that you do not have older registry entries from a previous installation, which may not be compatible with the latest version of Emacs.

When Emacs starts, as well as checking the environment, it also checks the System Registry for those variables and for `HOME`, `LANG` and `PRELOAD_WINSOCK`.

To determine the value of those variables, Emacs goes through the following procedure. First, the environment is checked. If the variable is not found there, Emacs looks for registry keys by that name under '`/Software/GNU/Emacs`'; first in the '`HKEY_CURRENT_USER`' section of the registry, and if not found there, in the '`HKEY_LOCAL_MACHINE`' section. Finally, if Emacs still cannot determine the values, compiled-in defaults are used.

In addition to the environment variables above, you can also add many of the settings which on X belong in the '`.Xdefaults`' file (see Appendix F [X Resources], page 769) to the '`/Software/GNU/Emacs`' registry key.

## E.5 Specifying the Display Name

The environment variable `DISPLAY` tells all X clients, including Emacs, where to display their windows. Its value is set by default in ordinary circumstances, when you start an X server and run jobs locally. You can specify the display yourself; one reason to do this is if you want to log into another system and run Emacs there, and have the window displayed at your local terminal.

`DISPLAY` has the syntax '*`host:display.screen`*', where *host* is the host name of the X Window System server machine, *display* is an arbitrarily-assigned number that distinguishes your server (X terminal) from other servers on the same machine, and *screen* is a field that allows an X server to control multiple terminal screens. The period and the *screen* field are optional. If included, *screen* is usually zero.

For example, if your host is named '`glasperle`' and your server is the first (or perhaps the only) server listed in the configuration, your `DISPLAY` is '`glasperle:0.0`'.

You can specify the display name explicitly when you run Emacs, either by changing the `DISPLAY` variable, or with the option '`-d` *`display`*' or '`--display=`*`display`*'. Here is an example:

```
emacs --display=glasperle:0 &
```

You can inhibit the use of the X window system with the '`-nw`' option. Then Emacs uses its controlling text terminal for display. See Section E.2 [Initial Options], page 755.

Sometimes, security arrangements prevent a program on a remote system from displaying on your local system. In this case, trying to run Emacs produces messages like this:

```
Xlib:  connection to "glasperle:0.0" refused by server
```

You might be able to overcome this problem by using the `xhost` command on the local system to give permission for access from your remote machine.

## E.6 Font Specification Options

You can use the command line option '`-fn` *`font`*' (or '`--font`', which is an alias for '`-fn`') to specify a default font:

'`-fn` *`font`*'
'`--font=`*`font`*'
: Use *font* as the default font.

When passing a font name to Emacs on the command line, you may need to "quote" it, by enclosing it in quotation marks, if it contains characters that the shell treats specially (e.g., spaces). For example:

```
emacs -fn "DejaVu Sans Mono-12"
```

See ⟨undefined⟩ [Fonts], page ⟨undefined⟩, for details about font names and other ways to specify the default font.

## E.7 Window Color Options

You can use the following command-line options to specify the colors to use for various parts of the Emacs display. Colors may be specified using either color names or RGB triplets (see ⟨undefined⟩ [Colors], page ⟨undefined⟩).

'`-fg` *`color`*'
'`--foreground-color=`*`color`*'
: Specify the foreground color, overriding the color specified by the `default` face (see Section 11.12 [Faces], page 137).

'`-bg` *`color`*'
'`--background-color=`*`color`*'
: Specify the background color, overriding the color specified by the `default` face.

‘`-bd` *color*’
‘`--border-color=`*color*’
: Specify the color of the border of the X window. This has no effect if Emacs is compiled with GTK+ support.

‘`-cr` *color*’
‘`--cursor-color=`*color*’
: Specify the color of the Emacs cursor which indicates where point is.

‘`-ms` *color*’
‘`--mouse-color=`*color*’
: Specify the color for the mouse cursor when the mouse is in the Emacs window.

‘`-r`’
‘`-rv`’
‘`--reverse-video`’
: Reverse video—swap the foreground and background colors.

‘`--color=`*mode*’
: Set the *color support mode* when Emacs is run on a text terminal. This option overrides the number of supported colors that the character terminal advertises in its `termcap` or `terminfo` database. The parameter *mode* can be one of the following:

  ‘`never`’
  ‘`no`’ — Don’t use colors even if the terminal’s capabilities specify color support.

  ‘`default`’
  ‘`auto`’ — Same as when ‘`--color`’ is not used at all: Emacs detects at startup whether the terminal supports colors, and if it does, turns on colored display.

  ‘`always`’
  ‘`yes`’
  ‘`ansi8`’ — Turn on the color support unconditionally, and use color commands specified by the ANSI escape sequences for the 8 standard colors.

  ‘*num*’ — Use color mode for *num* colors. If *num* is -1, turn off color support (equivalent to ‘`never`’); if it is 0, use the default color support for this terminal (equivalent to ‘`auto`’); otherwise use an appropriate standard mode for *num* colors. Depending on your terminal’s capabilities, Emacs might be able to turn on a color mode for 8, 16, 88, or 256 as the value of *num*. If there is no mode that supports *num* colors, Emacs acts as if *num* were 0, i.e., it uses the terminal’s default color support mode.

  If *mode* is omitted, it defaults to *ansi8*.

For example, to use a coral mouse cursor and a slate blue text cursor, enter:

```
emacs -ms coral -cr 'slate blue' &
```

You can reverse the foreground and background colors through the ‘`-rv`’ option or with the X resource ‘`reverseVideo`’.

The '`-fg`', '`-bg`', and '`-rv`' options function on text terminals as well as on graphical displays.

## E.8 Options for Window Size and Position

Here is a list of the command-line options for specifying size and position of the initial Emacs frame:

'`-g` *width*`x`*height*[`{+-}`*xoffset*`{+-}`*yoffset*]]'
'`--geometry=`*width*`x`*height*[`{+-}`*xoffset*`{+-}`*yoffset*]]'
: Specify the size *width* and *height* (measured in character columns and lines), and positions *xoffset* and *yoffset* (measured in pixels). The *width* and *height* parameters apply to all frames, whereas *xoffset* and *yoffset* only to the initial frame.

'`-fs`'
'`--fullscreen`'
: Specify that width and height shall be the size of the screen. Normally no window manager decorations are shown.

'`-mm`'
'`--maximized`'
: Specify that the Emacs frame shall be maximized. This normally means that the frame has window manager decorations.

'`-fh`'
'`--fullheight`'
: Specify that the height shall be the height of the screen.

'`-fw`'
'`--fullwidth`'
: Specify that the width shall be the width of the screen.

In the '`--geometry`' option, `{+-}` means either a plus sign or a minus sign. A plus sign before *xoffset* means it is the distance from the left side of the screen; a minus sign means it counts from the right side. A plus sign before *yoffset* means it is the distance from the top of the screen, and a minus sign there indicates the distance from the bottom. The values *xoffset* and *yoffset* may themselves be positive or negative, but that doesn't change their meaning, only their direction.

Emacs uses the same units as `xterm` does to interpret the geometry. The *width* and *height* are measured in characters, so a large font creates a larger frame than a small font. (If you specify a proportional font, Emacs uses its maximum bounds width as the width unit.) The *xoffset* and *yoffset* are measured in pixels.

You do not have to specify all of the fields in the geometry specification. If you omit both *xoffset* and *yoffset*, the window manager decides where to put the Emacs frame, possibly by letting you place it with the mouse. For example, '`164x55`' specifies a window 164 columns wide, enough for two ordinary width windows side by side, and 55 lines tall.

The default frame width is 80 characters and the default height is 40 lines. You can omit either the width or the height or both. If you start the geometry with an integer, Emacs

interprets it as the width. If you start with an '`x`' followed by an integer, Emacs interprets it as the height. Thus, '`81`' specifies just the width; '`x45`' specifies just the height.

If you start with '`+`' or '`-`', that introduces an offset, which means both sizes are omitted. Thus, '`-3`' specifies the *xoffset* only. (If you give just one offset, it is always *xoffset*.) '`+3-3`' specifies both the *xoffset* and the *yoffset*, placing the frame near the bottom left of the screen.

You can specify a default for any or all of the fields in your X resource file (see Section F.1 [Resources], page 769), and then override selected fields with a '`--geometry`' option.

Since the mode line and the echo area occupy the last 2 lines of the frame, the height of the initial text window is 2 less than the height specified in your geometry. In non-X-toolkit versions of Emacs, the menu bar also takes one line of the specified number. But in the X toolkit version, the menu bar is additional and does not count against the specified height. The tool bar, if present, is also additional.

Enabling or disabling the menu bar or tool bar alters the amount of space available for ordinary text. Therefore, if Emacs starts up with a tool bar (which is the default), and handles the geometry specification assuming there is a tool bar, and then your initialization file disables the tool bar, you will end up with a frame geometry different from what you asked for. To get the intended size with no tool bar, use an X resource to specify "no tool bar" (see Section F.2 [Table of Resources], page 770); then Emacs will already know there's no tool bar when it processes the specified geometry.

When using one of '`--fullscreen`', '`--maximized`', '`--fullwidth`' or '`--fullheight`' there may be some space around the frame anyway. That is because Emacs rounds the sizes so they are an even number of character heights and widths.

Some window managers have options that can make them ignore both program-specified and user-specified positions. If these are set, Emacs fails to position the window correctly.

## E.9 Internal and External Borders

An Emacs frame has an internal border and an external border. The internal border is an extra strip of the background color around the text portion of the frame. Emacs itself draws the internal border. The external border is added by the window manager outside the frame; depending on the window manager you use, it may contain various boxes you can click on to move or iconify the window.

'`-ib` *width*'
'`--internal-border=`*width*'
: Specify *width* as the width of the internal border (between the text and the main border), in pixels.

'`-bw` *width*'
'`--border-width=`*width*'
: Specify *width* as the width of the main border, in pixels.

When you specify the size of the frame, that does not count the borders. The frame's position is measured from the outside edge of the external border.

Use the '`-ib` *n*' option to specify an internal border *n* pixels wide. The default is 1. Use '`-bw` *n*' to specify the width of the external border (though the window manager may not pay attention to what you specify). The default width of the external border is 2.

## E.10 Frame Titles

An Emacs frame may or may not have a specified title. The frame title, if specified, appears in window decorations and icons as the name of the frame. If an Emacs frame has no specified title, the default title has the form '*invocation-name@machine*' (if there is only one frame) or the selected window's buffer name (if there is more than one frame).

You can specify a title for the initial Emacs frame with a command line option:

'`-T` *title*'
'`--title=`*title*'
: Specify *title* as the title for the initial Emacs frame.

The '`--name`' option (see Section F.1 [Resources], page 769) also specifies the title for the initial Emacs frame.

## E.11 Icons

'`-iconic`'
'`--iconic`'
: Start Emacs in an iconified ("minimized") state.

'`-nbi`'
'`--no-bitmap-icon`'
: Disable the use of the Emacs icon.

Most window managers allow you to "iconify" (or "minimize") an Emacs frame, hiding it from sight. Some window managers replace iconified windows with tiny "icons", while others remove them entirely from sight. The '`-iconic`' option tells Emacs to begin running in an iconified state, rather than showing a frame right away. The text frame doesn't appear until you deiconify (or "un-minimize") it.

By default, Emacs uses an icon containing the Emacs logo. On desktop environments such as Gnome, this icon is also displayed in other contexts, e.g., when switching into an Emacs frame. The '`-nbi`' or '`--no-bitmap-icon`' option tells Emacs to let the window manager choose what sort of icon to use—usually just a small rectangle containing the frame's title.

## E.12 Other Display Options

'`--parent-id` *ID*'
: Open Emacs as a client X window via the XEmbed protocol, with *ID* as the parent X window id. Currently, this option is mainly useful for developers.

'`-vb`'
'`--vertical-scroll-bars`'
: Enable vertical scroll bars.

'`-lsp` *pixels*'
'`--line-spacing=`*pixels*'
: Specify *pixels* as additional space to put between lines, in pixels.

‘`-nbc`’
‘`--no-blinking-cursor`’
: Disable the blinking cursor on graphical displays.

‘`-D`’
‘`--basic-display`’
: Disable the menu-bar, the tool-bar, the scroll-bars, and tool tips, and turn off the blinking cursor. This can be useful for making a test case that simplifies debugging of display problems.

The ‘`--xrm`’ option (see Section F.1 [Resources], page 769) specifies additional X resource values.

# Appendix F X Options and Resources

You can customize some X-related aspects of Emacs behavior using X resources, as is usual for programs that use X.

When Emacs is compiled with GTK+ support, the appearance of various graphical widgets, such as the menu-bar, scroll-bar, and dialog boxes, is determined by "GTK resources". When Emacs is built without GTK+ support, the appearance of these widgets is determined by additional X resources.

On MS-Windows, you can customize some of the same aspects using the system registry (see Section E.4.3 [MS-Windows Registry], page 762).

## F.1 X Resources

Programs running under the X Window System organize their user options under a hierarchy of classes and resources. You can specify default values for these options in your *X resource file*, usually named '`~/.Xdefaults`' or '`~/.Xresources`'. Changes in this file do not take effect immediately, because the X server stores its own list of resources; to update it, use the command `xrdb`—for instance, '`xrdb ~/.Xdefaults`'.

(MS-Windows systems do not support X resource files; on such systems, Emacs looks for X resources in the Windows Registry, first under the key '`HKEY_CURRENT_USER\SOFTWARE\GNU\Emacs`', which affects only the current user and override the system-wide settings, and then under the key '`HKEY_LOCAL_MACHINE\SOFTWARE\GNU\Emacs`', which affects all users of the system. The menu and scroll bars are native widgets on MS-Windows, so they are only customizable via the system-wide settings in the Display Control Panel. You can also set resources using the '`-xrm`' command line option, as explained below.)

Each line in the X resource file specifies a value for one option or for a collection of related options. The order in which the lines appear in the file does not matter. Each resource specification consists of a *program name* and a *resource name*. Case distinctions are significant in each of these names. Here is an example:

```
emacs.cursorColor: dark green
```

The program name is the name of the executable file to which the resource applies. For Emacs, this is normally '`emacs`'. To specify a definition that applies to all instances of Emacs, regardless of the name of the Emacs executable, use '`Emacs`'.

The resource name is the name of a program setting. For instance, Emacs recognizes a '`cursorColor`' resource that controls the color of the text cursor.

Resources are grouped into named classes. For instance, the '`Foreground`' class contains the '`cursorColor`', '`foreground`' and '`pointerColor`' resources (see Section F.2 [Table of Resources], page 770). Instead of using a resource name, you can use a class name to specify the default value for all resources in that class, like this:

```
emacs.Foreground: dark green
```

Emacs does not process X resources at all if you set the variable `inhibit-x-resources` to a non-`nil` value. If you invoke Emacs with the '`-Q`' (or '`--quick`') command-line option, `inhibit-x-resources` is automatically set to `t` (see Section E.2 [Initial Options], page 755).

## F.2 Table of X Resources for Emacs

This table lists the X resource names that Emacs recognizes, excluding those that control the appearance of graphical widgets like the menu bar:

`background` (class `Background`)
: Background color (see ⟨undefined⟩ [Colors], page ⟨undefined⟩).

`bitmapIcon` (class `BitmapIcon`)
: Tell the window manager to display the Emacs icon if '`on`'; don't do so if '`off`'. See Section E.11 [Icons X], page 767, for a description of the icon.

`cursorColor` (class `Foreground`)
: Text cursor color. If this resource is specified when Emacs starts up, Emacs sets its value as the background color of the `cursor` face (see Section 11.12 [Faces], page 137).

`cursorBlink` (class `CursorBlink`)
: If the value of this resource is '`off`' or '`false`' or '`0`' at startup, Emacs disables Blink Cursor mode (see ⟨undefined⟩ [Cursor Display], page ⟨undefined⟩).

`font` (class `Font`)
: Font name for the `default` face (see ⟨undefined⟩ [Fonts], page ⟨undefined⟩). You can also specify a fontset name (see Section 19.14 [Fontsets], page 389).

`fontBackend` (class `FontBackend`)
: Comma-delimited list of backend(s) to use for drawing fonts, in order of precedence. For instance, the value '`x,xft`' tells Emacs to draw fonts using the X core font driver, falling back on the Xft font driver if that fails. Normally, you should leave this resource unset, in which case Emacs tries using all available font backends.

`foreground` (class `Foreground`)
: Default foreground color for text.

`geometry` (class `Geometry`)
: Window size and position. The value should be a size and position specification, of the same form as in the '`-g`' or '`--geometry`' command-line option (see Section E.8 [Window Size X], page 765).

  The size applies to all frames in the Emacs session, but the position applies only to the initial Emacs frame (or, in the case of a resource for a specific frame name, only that frame).

  Be careful not to specify this resource as '`emacs*geometry`', as that may affect individual menus as well as the main Emacs frame.

`fullscreen` (class `Fullscreen`)
: The desired fullscreen size. The value can be one of `fullboth`, `maximized`, `fullwidth` or `fullheight`, which correspond to the command-line options '`-fs`', '`-mm`', '`-fw`', and '`-fh`' (see Section E.8 [Window Size X], page 765). Note that this applies to the initial frame only.

`lineSpacing` (class `LineSpacing`)
: Additional space between lines, in pixels.

`menuBar` (class `MenuBar`)
: If the value of this resource is '`off`' or '`false`' or '`0`', Emacs disables Menu Bar mode at startup (see ⟨undefined⟩ [Menu Bars], page ⟨undefined⟩).

`pointerColor` (class `Foreground`)
: Color of the mouse cursor. This has no effect in many graphical desktop environments, as they do not let Emacs change the mouse cursor this way.

`title` (class `Title`)
: Name to display in the title bar of the initial Emacs frame.

`toolBar` (class `ToolBar`)
: If the value of this resource is '`off`' or '`false`' or '`0`', Emacs disables Tool Bar mode at startup (see ⟨undefined⟩ [Tool Bars], page ⟨undefined⟩).

`useXIM` (class `UseXIM`)
: Disable use of X input methods (XIM) if '`false`' or '`off`'. This is only relevant if your Emacs is built with XIM support. It might be useful to turn off XIM on slow X client/server links.

`verticalScrollBars` (class `ScrollBars`)
: Give frames scroll bars if '`on`'; don't have scroll bars if '`off`'.

You can also use X resources to customize individual Emacs faces (see Section 11.12 [Faces], page 137). For example, setting the resource '*`face`*`.attributeForeground`' is equivalent to customizing the '`foreground`' attribute of the face *face*. However, we recommend customizing faces from within Emacs, instead of using X resources. See Section 33.1.5 [Face Customization], page 690.

## F.3 GTK resources

If Emacs is compiled with GTK+ toolkit support, the simplest way to customize its GTK+ widgets (e.g., menus, dialogs, tool bars and scroll bars) is to choose an appropriate GTK+ theme, for example with the GNOME theme selector.

In GTK+ version 2, you can also use *GTK+ resources* to customize the appearance of GTK+ widgets used by Emacs. These resources are specified in either the file '`~/.emacs.d/gtkrc`' (for Emacs-specific GTK+ resources), or '`~/.gtkrc-2.0`' (for general GTK+ resources). We recommend using '`~/.emacs.d/gtkrc`', since GTK+ seems to ignore '`~/.gtkrc-2.0`' when running GConf with GNOME. Note, however, that some GTK themes may override customizations in '`~/.emacs.d/gtkrc`'; there is nothing we can do about this. GTK+ resources do not affect aspects of Emacs unrelated to GTK+ widgets, such as fonts and colors in the main Emacs window; those are governed by normal X resources (see Section F.1 [Resources], page 769).

The following sections describe how to customize GTK+ resources for Emacs. For details about GTK+ resources, see the GTK+ API document at `http://developer.gnome.org/gtk2/stable/gtk2-Resource-Files.html`.

In GTK+ version 3, GTK+ resources have been replaced by a completely different system. The appearance of GTK+ widgets is now determined by CSS-like style files: '`gtk-3.0/gtk.css`' in the GTK+ installation directory, and '`~/.themes/`*`theme`*`/gtk-3.0/gtk.css`' for local style settings (where *theme* is the name

of the current GTK+ theme). Therefore, the description of GTK+ resources in this section does not apply to GTK+ 3. For details about the GTK+ 3 styling system, see http://developer.gnome.org/gtk3/3.0/GtkCssProvider.html.

### F.3.1 GTK Resource Basics

In a GTK+ 2 resource file (usually '`~/.emacs.d/gtkrc`'), the simplest kinds of resource settings simply assign a value to a variable. For example, putting the following line in the resource file changes the font on all GTK+ widgets to '`courier-12`':

```
gtk-font-name = "courier 12"
```

Note that in this case the font name must be supplied as a GTK font pattern (also called a *Pango font name*), not as a Fontconfig-style font name or XLFD. See ⟨undefined⟩ [Fonts], page ⟨undefined⟩.

To customize widgets you first define a *style*, and then apply the style to the widgets. Here is an example that sets the font for menus ('`#`' characters indicate comments):

```
# Define the style 'my_style'.
style "my_style"
{
  font_name = "helvetica bold 14"
}

# Specify that widget type '*emacs-menuitem*' uses 'my_style'.
widget "*emacs-menuitem*" style "my_style"
```

The widget name in this example contains wildcards, so the style is applied to all widgets matching '`*emacs-menuitem*`'. The widgets are named by the way they are contained, from the outer widget to the inner widget. Here is another example that applies '`my_style`' specifically to the Emacs menu bar:

```
widget "Emacs.pane.menubar.*" style "my_style"
```

Here is a more elaborate example, showing how to change the parts of the scroll bar:

```
style "scroll"
{
  fg[NORMAL] = "red"      # Arrow color.
  bg[NORMAL] = "yellow"   # Thumb and background around arrow.
  bg[ACTIVE] = "blue"     # Trough color.
  bg[PRELIGHT] = "white" # Thumb color when the mouse is over it.
}

widget "*verticalScrollBar*" style "scroll"
```

### F.3.2 GTK widget names

A GTK+ widget is specified by a *widget name* and a *widget class*. The widget name refers to a specific widget (e.g., '`emacs-menuitem`'), while the widget class refers to a collection of similar widgets (e.g., '`GtkMenuItem`'). A widget always has a class, but need not have a name.

*Absolute names* are sequences of widget names or widget classes, corresponding to hierarchies of widgets embedded within other widgets. For example, if a `GtkWindow` named `top` contains a `GtkVBox` named `box`, which in turn contains a `GtkMenuBar` called `menubar`, the absolute class name of the menu-bar widget is `GtkWindow.GtkVBox.GtkMenuBar`, and its absolute widget name is `top.box.menubar`.

GTK+ resource files can contain two types of commands for specifying widget appearances:

`widget` specifies a style for widgets based on the class name, or just the class.

`widget_class`
specifies a style for widgets based on the class name.

See the previous subsection for examples of using the `widget` command; the `widget_class` command is used similarly. Note that the widget name/class and the style must be enclosed in double-quotes, and these commands must be at the top level in the GTK+ resource file.

As previously noted, you may specify a widget name or class with shell wildcard syntax: '`*`' matches zero or more characters and '`?`' matches one character. This example assigns a style to all widgets:

```
widget "*" style "my_style"
```

### F.3.3 GTK Widget Names in Emacs

The GTK+ widgets used by an Emacs frame are listed below:

`Emacs` (class `GtkWindow`)
`pane` (class `GtkVBox`)
`menubar` (class `GtkMenuBar`)
[menu item widgets]
[unnamed widget] (class `GtkHandleBox`)
`emacs-toolbar` (class `GtkToolbar`)
[tool bar item widgets]
`emacs` (class `GtkFixed`)
`verticalScrollBar` (class `GtkVScrollbar`)

The contents of Emacs windows are drawn in the `emacs` widget. Note that even if there are multiple Emacs windows, each scroll bar widget is named `verticalScrollBar`.

For example, here are two different ways to set the menu bar style:

```
widget "Emacs.pane.menubar.*" style "my_style"
widget_class "GtkWindow.GtkVBox.GtkMenuBar.*" style "my_style"
```

For GTK+ dialogs, Emacs uses a widget named `emacs-dialog`, of class `GtkDialog`. For file selection, Emacs uses a widget named `emacs-filedialog`, of class `GtkFileSelection`.

Because the widgets for pop-up menus and dialogs are free-standing windows and not "contained" in the `Emacs` widget, their GTK+ absolute names do not start with '`Emacs`'. To customize these widgets, use wildcards like this:

```
widget "*emacs-dialog*" style "my_dialog_style"
widget "*emacs-filedialog* style "my_file_style"
widget "*emacs-menuitem* style "my_menu_style"
```

If you want to apply a style to all menus in Emacs, use this:

```
widget_class "*Menu*" style "my_menu_style"
```

### F.3.4 GTK styles

Here is an example of two GTK+ style declarations:

```
pixmap_path "/usr/share/pixmaps:/usr/include/X11/pixmaps"

style "default"
{
  font_name = "helvetica 12"

  bg[NORMAL] = { 0.83, 0.80, 0.73 }
  bg[SELECTED] = { 0.0, 0.55, 0.55 }
  bg[INSENSITIVE] = { 0.77, 0.77, 0.66 }
  bg[ACTIVE] = { 0.0, 0.55, 0.55 }
  bg[PRELIGHT] = { 0.0, 0.55, 0.55 }

  fg[NORMAL] = "black"
  fg[SELECTED] = { 0.9, 0.9, 0.9 }
  fg[ACTIVE] = "black"
  fg[PRELIGHT] = { 0.9, 0.9, 0.9 }

  base[INSENSITIVE] = "#777766"
  text[INSENSITIVE] = { 0.60, 0.65, 0.57 }

  bg_pixmap[NORMAL] = "background.xpm"
  bg_pixmap[INSENSITIVE] = "background.xpm"
  bg_pixmap[ACTIVE] = "background.xpm"
  bg_pixmap[PRELIGHT] = "<none>"

}

style "ruler" = "default"
{
  font_name = "helvetica 8"
}
```

The style '`ruler`' inherits from '`default`'. This way you can build on existing styles. The syntax for fonts and colors is described below.

As this example shows, it is possible to specify several values for foreground and background depending on the widget's *state*. The possible states are:

`NORMAL` This is the default state for widgets.

`ACTIVE` This is the state for a widget that is ready to do something. It is also for the trough of a scroll bar, i.e., `bg[ACTIVE] = "red"` sets the scroll bar trough to red. Buttons that have been pressed but not released yet ("armed") are in this state.

`PRELIGHT` This is the state for a widget that can be manipulated, when the mouse pointer is over it—for example when the mouse is over the thumb in the scroll bar or over a menu item. When the mouse is over a button that is not pressed, the button is in this state.

`SELECTED` This is the state for data that has been selected by the user. It can be selected text or items selected in a list. This state is not used in Emacs.

`INSENSITIVE`
: This is the state for widgets that are visible, but they can not be manipulated in the usual way—for example, buttons that can't be pressed, and disabled menu items. To display disabled menu items in yellow, use `fg[INSENSITIVE] = "yellow"`.

Here are the things that can go in a style declaration:

`bg[`*state*`] = `*color*
: This specifies the background color for the widget. Note that editable text doesn't use `bg`; it uses `base` instead.

`base[`*state*`] = `*color*
: This specifies the background color for editable text. In Emacs, this color is used for the background of the text fields in the file dialog.

`bg_pixmap[`*state*`] = "`*pixmap*`"`
: This specifies an image background (instead of a background color). *pixmap* should be the image file name. GTK can use a number of image file formats, including XPM, XBM, GIF, JPEG and PNG. If you want a widget to use the same image as its parent, use '`<parent>`'. If you don't want any image, use '`<none>`'. '`<none>`' is the way to cancel a background image inherited from a parent style.

  You can't specify the file by its absolute file name. GTK looks for the pixmap file in directories specified in `pixmap_path`. `pixmap_path` is a colon-separated list of directories within double quotes, specified at the top level in a '`gtkrc`' file (i.e., not inside a style definition; see example above):

```
pixmap_path "/usr/share/pixmaps:/usr/include/X11/pixmaps"
```

`fg[`*state*`] = `*color*
: This specifies the foreground color for widgets to use. It is the color of text in menus and buttons, and the color for the arrows in the scroll bar. For editable text, use `text`.

`text[`*state*`] = `*color*
: This is the color for editable text. In Emacs, this color is used for the text fields in the file dialog.

`font_name = "`*font*`"`
: This specifies the font for text in the widget. *font* is a GTK-style (or Pango) font name, like '`Sans Italic 10`'. See ⟨undefined⟩ [Fonts], page ⟨undefined⟩. The names are case insensitive.

There are three ways to specify a color: a color name, an RGB triplet, or a GTK-style RGB triplet. See ⟨undefined⟩ [Colors], page ⟨undefined⟩, for a description of color names and RGB triplets. Color names should be enclosed with double quotes, e.g., '`"red"`'. RGB triplets should be written without double quotes, e.g., '`#ff0000`'. GTK-style RGB triplets have the form `{ r, g, b }`, where *r*, *g* and *b* are either integers in the range 0–65535 or floats in the range 0.0–1.0.

# Appendix G Emacs 23 Antinews

For those users who live backwards in time, here is information about downgrading to Emacs version 23.4. We hope you will enjoy the greater simplicity that results from the absence of many Emacs 24.3 features.

## G.1 Old Lisp Features in Emacs 23

- Support for lexical scoping has been removed; all variables are dynamically scoped. The `lexical-binding` variable has been removed, and so has the *lexical* argument to `eval`. The `defvar` and `defconst` forms no longer mark variables as dynamic, since all variables are dynamic.

  Having only dynamic binding follows the spirit of Emacs extensibility, for it allows any Emacs code to access any defined variable with a minimum of fuss. But See ⟨undefined⟩ [Dynamic Binding Tips], page ⟨undefined⟩, for tips to avoid making your programs hard to understand.
- Calling a minor mode function from Lisp with a nil or omitted argument does not enable the minor mode unconditionally; instead, it toggles the minor mode—which is the straightforward thing to do, since that is the behavior when invoked interactively. One downside is that it is more troublesome to enable minor modes from hooks; you have to do something like

  ```
  (add-hook 'foo-hook (lambda () (bar-mode 1)))
  ```

  or define `turn-on-bar-mode` and call that from the hook.
- The `prog-mode` dummy major mode has been removed. Instead of using it as a crutch to meet programming mode conventions, you should explicitly ensure that your mode follows those conventions. See Section 20.2.1 [Major Mode Conventions], page 399.
- Emacs no longer supports bidirectional display and editing. Since there is no need to worry about the insertion of right-to-left text messing up how lines and paragraphs are displayed, the function `bidi-string-mark-left-to-right` has been removed; so have many other functions and variables related to bidirectional display. Unicode directionality characters like `U+200E` ("left-to-right mark") have no special effect on display.
- Emacs windows now have most of their internal state hidden from Lisp. Internal windows are no longer visible to Lisp; functions such as `window-parent`, window parameters related to window arrangement, and window-local buffer lists have all been removed. Functions for resizing windows can delete windows if they become too small.

  The "action function" feature for controlling buffer display has been removed, including `display-buffer-overriding-action` and related variables, as well as the *action* argument to `display-buffer` and other functions. The way to programmatically control how Emacs chooses a window to display a buffer is to bind the right combination of `pop-up-frames` and other variables.
- The standard completion interface has been simplified, eliminating the `completion-extra-properties` variable, the `metadata` action flag for completion functions, and the concept of "completion categories". Lisp programmers may now find the choice of methods for tuning completion less bewildering, but if a package finds the streamlined

interface insufficient for its needs, it must implement its own specialized completion feature.

- `copy-directory` now behaves the same whether or not the destination is an existing directory: if the destination exists, the *contents* of the first directory are copied into it (with subdirectories handled recursively), rather than copying the first directory into a subdirectory.
- The *trash* arguments for `delete-file` and `delete-directory` have been removed. The variable `delete-by-moving-to-trash` must now be used with care; whenever it is non-`nil`, all calls to `delete-file` or `delete-directory` use the trash.
- Because Emacs no longer supports SELinux file contexts, the *preserve-selinux-context* argument to `copy-file` has been removed. The return value of `backup-buffer` no longer has an entry for the SELinux file context.
- For mouse click input events in the text area, the Y pixel coordinate in the *position* list (see Section 2.7.4 [Click Events], page 26) now counts from the top of the header line, if there is one, rather than the top of the text area.
- Bindings in menu keymaps (see ⟨undefined⟩ [Format of Keymaps], page ⟨undefined⟩) now sometimes get an additional *cache* entry in their definitions, like this:

  ```
  (type item-name cache . binding)
  ```

  The *cache* entry is used internally by Emacs to record equivalent keyboard key sequences for invoking the same command; Lisp programs should never use it.
- The `gnutls` library has been removed, and the function `open-network-stream` correspondingly simplified. Lisp programs that want an encrypted network connection must now call external utilities such as `starttls` or `gnutls-cli`.
- Tool bars can no longer display separators, which frees up several pixels of space on each graphical frame.
- As part of the ongoing quest for simplicity, many other functions and variables have been eliminated.

# Appendix H Emacs and Mac OS / GNUstep

This section describes the peculiarities of using Emacs built with the GNUstep libraries on GNU/Linux or other operating systems, or on Mac OS X with native window system support. On Mac OS X, Emacs can be built either without window system support, with X11, or with the Cocoa interface; this section only applies to the Cocoa build. This does not support versions of Mac OS X earlier than 10.4.

For various historical and technical reasons, Emacs uses the term '`Nextstep`' internally, instead of "Cocoa" or "Mac OS X"; for instance, most of the commands and variables described in this section begin with '`ns-`', which is short for '`Nextstep`'. NeXTstep was an application interface released by NeXT Inc during the 1980s, of which Cocoa is a direct descendant. Apart from Cocoa, there is another NeXTstep-style system: GNUstep, which is free software. As of this writing, Emacs GNUstep support is alpha status (see Section H.4 [GNUstep Support], page 780), but we hope to improve it in the future.

## H.1 Basic Emacs usage under Mac OS and GNUstep

By default, the ALT and OPTION keys are the same as META. The Mac CMD key is the same as SUPER, and Emacs provides a set of key bindings using this modifier key that mimic other Mac / GNUstep applications (see Section H.3 [Mac / GNUstep Events], page 779). You can change these bindings in the usual way (see Section 33.3 [Key Bindings], page 702).

The variable `ns-right-alternate-modifier` controls the behavior of the right ALT and OPTION keys. These keys behave like the left-hand keys if the value is `left` (the default). A value of `control`, `meta`, `alt`, `super`, or `hyper` makes them behave like the corresponding modifier keys; a value of `none` tells Emacs to ignore them.

`S-Mouse-1` adjusts the region to the click position, just like `Mouse-3` (`mouse-save-then-kill`); it does not pop up a menu for changing the default face, as `S-Mouse-1` normally does (see ⟨undefined⟩ [Text Scale], page ⟨undefined⟩). This change makes Emacs behave more like other Mac / GNUstep applications.

When you open or save files using the menus, or using the `Cmd-o` and `Cmd-S` bindings, Emacs uses graphical file dialogs to read file names. However, if you use the regular Emacs key sequences, such as `C-x C-f`, Emacs uses the minibuffer to read file names.

On GNUstep, in an X-windows environment you need to use `Cmd-c` instead of one of the `C-w` or `M-w` commands to transfer text to the X primary selection; otherwise, Emacs will use the "clipboard" selection. Likewise, `Cmd-y` (instead of `C-y`) yanks from the X primary selection instead of the kill-ring or clipboard.

### H.1.1 Grabbing environment variables

Many programs which may run under Emacs, like latex or man, depend on the settings of environment variables. If Emacs is launched from the shell, it will automatically inherit these environment variables and its subprocesses will inherit them from it. But if Emacs is launched from the Finder it is not a descendant of any shell, so its environment variables haven't been set, which often causes the subprocesses it launches to behave differently than they would when launched from the shell.

For the PATH and MANPATH variables, a system-wide method of setting PATH is recommended on Mac OS X 10.5 and later, using the '`/etc/paths`' files and the '`/etc/paths.d`' directory.

## H.2 Mac / GNUstep Customization

Emacs can be customized in several ways in addition to the standard customization buffers and the Options menu.

### H.2.1 Font and Color Panels

The standard Mac / GNUstep font and color panels are accessible via Lisp commands. The Font Panel may be accessed with `M-x ns-popup-font-panel`. It will set the default font in the frame most recently used or clicked on.

You can bring up a color panel with `M-x ns-popup-color-panel` and drag the color you want over the Emacs face you want to change. Normal dragging will alter the foreground color. Shift dragging will alter the background color. To discard the settings, create a new frame and close the altered one.

Useful in this context is the listing of all faces obtained by `M-x list-faces-display`.

### H.2.2 Customization options specific to Mac OS / GNUstep

The following customization options are specific to the Nextstep port.

`ns-auto-hide-menu-bar`
: Non-nil means the menu-bar is hidden by default, but appears if you move the mouse pointer over it. (Requires Mac OS X 10.6 or later.)

## H.3 Windowing System Events under Mac OS / GNUstep

Nextstep applications receive a number of special events which have no X equivalent. These are sent as specially defined "keys", which do not correspond to any sequence of keystrokes. Under Emacs, these "key" events can be bound to functions just like ordinary keystrokes. Here is a list of these events.

NS-OPEN-FILE
: This event occurs when another Nextstep application requests that Emacs open a file. A typical reason for this would be a user double-clicking a file in the Finder application. By default, Emacs responds to this event by opening a new frame and visiting the file in that frame (`ns-find-file`). As an exception, if the selected buffer is the '`*scratch*`' buffer, Emacs visits the file in the selected frame.

  You can change how Emacs responds to a `ns-open-file` event by changing the variable `ns-pop-up-frames`. Its default value, '`fresh`', is what we have just described. A value of `t` means to always visit the file in a new frame. A value of `nil` means to always visit the file in an existing frame.

NS-OPEN-TEMP-FILE
: This event occurs when another application requests that Emacs open a temporary file. By default, this is handled by just generating a `ns-open-file` event, the results of which are described above.

NS-OPEN-FILE-LINE
: Some applications, such as ProjectBuilder and gdb, request not only a particular file, but also a particular line or sequence of lines in the file. Emacs handles this by visiting that file and highlighting the requested line (`ns-open-file-select-line`).

NS-DRAG-FILE
: This event occurs when a user drags files from another application into an Emacs frame. The default behavior is to insert the contents of all the dragged files into the current buffer (`ns-insert-files`). The list of dragged files is stored in the variable `ns-input-file`.

NS-DRAG-COLOR
: This event occurs when a user drags a color from the color well (or some other source) into an Emacs frame. The default behavior is to alter the foreground color of the area the color was dragged onto (`ns-set-foreground-at-mouse`). If this event is issued with a SHIFT modifier, Emacs changes the background color instead (`ns-set-background-at-mouse`). The name of the dragged color is stored in the variable `ns-input-color`.

NS-CHANGE-FONT
: This event occurs when the user selects a font in a Nextstep font panel (which can be opened with `Cmd-t`). The default behavior is to adjust the font of the selected frame (`ns-respond-to-changefont`). The name and size of the selected font are stored in the variables `ns-input-font` and `ns-input-fontsize`, respectively.

NS-POWER-OFF
: This event occurs when the user logs out and Emacs is still running, or when 'Quit Emacs' is chosen from the application menu. The default behavior is to save all file-visiting buffers.

Emacs also allows users to make use of Nextstep services, via a set of commands whose names begin with '`ns-service-`' and end with the name of the service. Type `M-x ns-service-TAB` to see a list of these commands. These functions either operate on marked text (replacing it with the result) or take a string argument and return the result as a string. You can also use the Lisp function `ns-perform-service` to pass arbitrary strings to arbitrary services and receive the results back. Note that you may need to restart Emacs to access newly-available services.

## H.4 GNUstep Support

Emacs can be built and run under GNUstep, but there are still issues to be addressed. Interested developers should contact `emacs-devel@gnu.org`.

# Appendix I Emacs and Microsoft Windows/MS-DOS

This section describes peculiarities of using Emacs on Microsoft Windows. Some of these peculiarities are also relevant to Microsoft's older MS-DOS "operating system" (also known as "MS-DOG"). However, Emacs features that are relevant *only* to MS-DOS are described in a separate manual (see Section "MS-DOS" in *Specialized Emacs Features*).

The behavior of Emacs on MS-Windows is reasonably similar to what is documented in the rest of the manual, including support for long file names, multiple frames, scroll bars, mouse menus, and subprocesses. However, a few special considerations apply, and they are described here.

## I.1 How to Start Emacs on MS-Windows

There are several ways of starting Emacs on MS-Windows:

1. From the desktop shortcut icon: either double-click the left mouse button on the icon, or click once, then press RET. The desktop shortcut should specify as its "Target" (in the "Properties" of the shortcut) the full absolute file name of '`runemacs.exe`', *not* of '`emacs.exe`'. This is because '`runemacs.exe`' hides the console window that would have been created if the target of the shortcut were '`emacs.exe`' (which is a console program, as far as Windows is concerned). If you use this method, Emacs starts in the directory specified by the shortcut. To control where that is, right-click on the shortcut, select "Properties", and in the "Shortcut" tab modify the "Start in" field to your liking.
2. From the Command Prompt window, by typing `emacs RET` at the prompt. The Command Prompt window where you did that will not be available for invoking other commands until Emacs exits. In this case, Emacs will start in the current directory of the Windows shell.
3. From the Command Prompt window, by typing `runemacs RET` at the prompt. The Command Prompt window where you did that will be immediately available for invoking other commands. In this case, Emacs will start in the current directory of the Windows shell.
4. Via '`emacsclient.exe`' or '`emacsclientw.exe`', which allow you to invoke Emacs from other programs, and to reuse a running Emacs process for serving editing jobs required by other programs. See Section 31.4 [Emacs Server], page 665. The difference between '`emacsclient.exe`' and '`emacsclientw.exe`' is that the former is a console program, while the latter is a Windows GUI program. Both programs wait for Emacs to signal that the editing job is finished, before they exit and return control to the program that invoked them. Which one of them to use in each case depends on the expectations of the program that needs editing services. If that program is itself a console (text-mode) program, you should use '`emacsclient.exe`', so that any of its messages and prompts appear in the same command window as those of the invoking program. By contrast, if the invoking program is a GUI program, you will be better off using '`emacsclientw.exe`', because '`emacsclient.exe`' will pop up a command window if it is invoked from a GUI program. A notable situation where you would want '`emacsclientw.exe`' is when you right-click on a file in the Windows Explorer and select "Open With" from the pop-up menu. Use the '`--alternate-editor=`' or '`-a`' options if Emacs might not be running

(or not running as a server) when `emacsclient` is invoked—that will always give you an editor. When invoked via `emacsclient`, Emacs will start in the current directory of the program that invoked `emacsclient`.

Note that, due to limitations of MS-Windows, Emacs cannot have both GUI and text-mode frames in the same session. It also cannot open text-mode frames on more than a single *Command Prompt* window, because each Windows program can have only one console at any given time. For these reasons, if you invoke `emacsclient` with the '`-c`' option, and the Emacs server runs in a text-mode session, Emacs will always create a new text-mode frame in the same *Command Prompt* window where it was started; a GUI frame will be created only if the server runs in a GUI session. Similarly, if you invoke `emacsclient` with the '`-t`' option, Emacs will create a GUI frame if the server runs in a GUI session, or a text-mode frame when the session runs in text mode in a *Command Prompt* window. See Section 31.4.2 [emacsclient Options], page 667.

## I.2 Text Files and Binary Files

GNU Emacs uses newline characters to separate text lines. This is the convention used on GNU, Unix, and other Posix-compliant systems.

By contrast, MS-DOS and MS-Windows normally use carriage-return linefeed, a two-character sequence, to separate text lines. (Linefeed is the same character as newline.) Therefore, convenient editing of typical files with Emacs requires conversion of these end-of-line (EOL) sequences. And that is what Emacs normally does: it converts carriage-return linefeed into newline when reading files, and converts newline into carriage-return linefeed when writing files. The same mechanism that handles conversion of international character codes does this conversion also (see Section 19.6 [Coding Systems], page 381).

One consequence of this special format-conversion of most files is that character positions as reported by Emacs (see Section 4.9 [Position Info], page 64) do not agree with the file size information known to the operating system.

In addition, if Emacs recognizes from a file's contents that it uses newline rather than carriage-return linefeed as its line separator, it does not perform EOL conversion when reading or writing that file. Thus, you can read and edit files from GNU and Unix systems on MS-DOS with no special effort, and they will retain their Unix-style end-of-line convention after you edit them.

The mode line indicates whether end-of-line translation was used for the current buffer. If MS-DOS end-of-line translation is in use for the buffer, the MS-Windows build of Emacs displays a backslash '\' after the coding system mnemonic near the beginning of the mode line (see Section 1.3 [Mode Line], page 8). If no EOL translation was performed, the string '`(Unix)`' is displayed instead of the backslash, to alert you that the file's EOL format is not the usual carriage-return linefeed.

To visit a file and specify whether it uses DOS-style or Unix-style end-of-line, specify a coding system (see Section 19.10 [Text Coding], page 386). For example, `C-x RET c unix RET C-x C-f foobar.txt` visits the file '`foobar.txt`' without converting the EOLs; if some line ends with a carriage-return linefeed pair, Emacs will display '`^M`' at the end of that line. Similarly, you can direct Emacs to save a buffer in a specified EOL format with the `C-x RET f` command. For example, to save a buffer with Unix EOL format, type `C-x RET f unix`

`RET C-x C-s`. If you visit a file with DOS EOL conversion, then save it with Unix EOL format, that effectively converts the file to Unix EOL style, like the `dos2unix` program.

When you use NFS, Samba, or some other similar method to access file systems that reside on computers using GNU or Unix systems, Emacs should not perform end-of-line translation on any files in these file systems—not even when you create a new file. To request this, designate these file systems as *untranslated* file systems by calling the function `add-untranslated-filesystem`. It takes one argument: the file system name, including a drive letter and optionally a directory. For example,

```
(add-untranslated-filesystem "Z:")
```

designates drive Z as an untranslated file system, and

```
(add-untranslated-filesystem "Z:\\foo")
```

designates directory '`\foo`' on drive Z as an untranslated file system.

Most often you would use `add-untranslated-filesystem` in your '`.emacs`' file, or in '`site-start.el`' so that all the users at your site get the benefit of it.

To countermand the effect of `add-untranslated-filesystem`, use the function `remove-untranslated-filesystem`. This function takes one argument, which should be a string just like the one that was used previously with `add-untranslated-filesystem`.

Designating a file system as untranslated does not affect character set conversion, only end-of-line conversion. Essentially, it directs Emacs to create new files with the Unix-style convention of using newline at the end of a line. See Section 19.6 [Coding Systems], page 381.

Some kinds of files should not be converted at all, because their contents are not really text. Therefore, Emacs on MS-Windows distinguishes certain files as *binary files*. (This distinction is not part of MS-Windows; it is made by Emacs only.) Binary files include executable programs, compressed archives, etc. Emacs uses the file name to decide whether to treat a file as binary: the variable `file-name-buffer-file-type-alist` defines the file-name patterns that indicate binary files. If a file name matches one of the patterns for binary files (those whose associations are of the type `(`*`pattern`*` . t)`, Emacs reads and writes that file using the `no-conversion` coding system (see Section 19.6 [Coding Systems], page 381) which turns off *all* coding-system conversions, not only the EOL conversion. `file-name-buffer-file-type-alist` also includes file-name patterns for files which are known to be Windows-style text files with carriage-return linefeed EOL format, such as '`CONFIG.SYS`'; Emacs always writes those files with Windows-style EOLs.

If a file that belongs to an untranslated file system matches one of the file-name patterns in `file-name-buffer-file-type-alist`, the EOL conversion is determined by `file-name-buffer-file-type-alist`.

## I.3 File Names on MS-Windows

MS-Windows and MS-DOS normally use a backslash, '`\`', to separate name units within a file name, instead of the slash used on other systems. Emacs on MS-DOS/MS-Windows permits use of either slash or backslash, and also knows about drive letters in file names.

On MS-DOS/MS-Windows, file names are case-insensitive, so Emacs by default ignores letter-case in file names during completion.

The variable `w32-get-true-file-attributes` controls whether Emacs should issue additional system calls to determine more accurately file attributes in primitives like `file-`

`attributes` and `directory-files-and-attributes`. These additional calls are needed to report correct file ownership, link counts and file types for special files such as pipes. Without these system calls, file ownership will be attributed to the current user, link counts will be always reported as 1, and special files will be reported as regular files.

If the value of this variable is `local` (the default), Emacs will issue these additional system calls only for files on local fixed drives. Any other non-`nil` value means do this even for removable and remote volumes, where this could potentially slow down Dired and other related features. The value of `nil` means never issue those system calls. Non-`nil` values are more useful on NTFS volumes, which support hard links and file security, than on FAT, FAT32, and XFAT volumes.

## I.4 Emulation of `ls` on MS-Windows

Dired normally uses the external program `ls` to produce the directory listing displayed in Dired buffers (see Chapter 27 [Dired], page 588). However, MS-Windows and MS-DOS systems don't come with such a program, although several ports of GNU `ls` are available. Therefore, Emacs on those systems *emulates* `ls` in Lisp, by using the '`ls-lisp.el`' package. While '`ls-lisp.el`' provides a reasonably full emulation of `ls`, there are some options and features peculiar to that emulation; for more details, see the documentation of the variables whose names begin with `ls-lisp`.

## I.5 HOME and Startup Directories on MS-Windows

The Windows equivalent of `HOME` is the *user-specific application data directory*. The actual location depends on the Windows version; typical values are '`C:\Documents and Settings\`*username*`\Application Data`' on Windows 2000/XP/2K3, '`C:\Users\`*username*`\AppData\Roaming`' on Windows Vista/7/2008, and either '`C:\WINDOWS\Application Data`' or '`C:\WINDOWS\Profiles\`*username*`\Application Data`' on Windows 9X/ME. If this directory does not exist or cannot be accessed, Emacs falls back to '`C:\`' as the default value of `HOME`.

You can override this default value of `HOME` by explicitly setting the environment variable `HOME` to point to any directory on your system. `HOME` can be set either from the command shell prompt or from '`Properties`' dialog of '`My Computer`'. `HOME` can also be set in the system registry, see Section E.4.3 [MS-Windows Registry], page 762.

For compatibility with older versions of Emacs[1], if there is a file named '`.emacs`' in '`C:\`', the root directory of drive '`C:`', and `HOME` is set neither in the environment nor in the Registry, Emacs will treat '`C:\`' as the default `HOME` location, and will not look in the application data directory, even if it exists. Note that only '`.emacs`' is looked for in '`C:\`'; the older name '`_emacs`' (see below) is not. This use of '`C:\.emacs`' to define `HOME` is deprecated.

Whatever the final place is, Emacs sets the internal value of the `HOME` environment variable to point to it, and it will use that location for other files and directories it normally looks for or creates in your home directory.

You can always find out what Emacs thinks is your home directory's location by typing `C-x d ~/ RET`. This should present the list of files in the home directory, and show its full

[1] Older versions of Emacs didn't check the application data directory.

name on the first line. Likewise, to visit your init file, type `C-x C-f ~/.emacs RET` (assuming the file's name is '`.emacs`').

The home directory is where your init file is stored. It can have any name mentioned in Section 33.4 [Init File], page 711.

Because MS-DOS does not allow file names with leading dots, and older Windows systems made it hard to create files with such names, the Windows port of Emacs supports an init file name '`_emacs`', if such a file exists in the home directory and '`.emacs`' does not. This name is considered obsolete.

## I.6 Keyboard Usage on MS-Windows

This section describes the Windows-specific features related to keyboard input in Emacs.

Many key combinations (known as "keyboard shortcuts") that have conventional uses in MS-Windows programs conflict with traditional Emacs key bindings. (These Emacs key bindings were established years before Microsoft was founded.) Examples of conflicts include `C-c`, `C-x`, `C-z`, `C-a`, and `W-SPC`. You can redefine some of them with meanings more like the MS-Windows meanings by enabling CUA Mode (see Section 9.6 [CUA Bindings], page 105).

The F10 key on Windows activates the menu bar in a way that makes it possible to use the menus without a mouse. In this mode, the arrow keys traverse the menus, RET selects a highlighted menu item, and ESC closes the menu.

See Info file '`emacs`', node '`Windows Keyboard`', for information about additional Windows-specific variables in this category.

The variable `w32-apps-modifier` controls the effect of the APPS key (usually located between the right ALT and the right CTRL keys). Its value can be one of the symbols `hyper`, `super`, `meta`, `alt`, `control`, or `shift` for the respective modifier, or `nil` to appear as the key `apps`. The default is `nil`.

The variable `w32-lwindow-modifier` determines the effect of the left Windows key (usually labeled with START and the Windows logo). If its value is `nil` (the default), the key will produce the symbol `lwindow`. Setting it to one of the symbols `hyper`, `super`, `meta`, `alt`, `control`, or `shift` will produce the respective modifier. A similar variable `w32-rwindow-modifier` controls the effect of the right Windows key, and `w32-scroll-lock-modifier` does the same for the SCRLOCK key. If these variables are set to `nil`, the right Windows key produces the symbol `rwindow` and SCRLOCK produces the symbol `scroll`.

Emacs compiled as a native Windows application normally turns off the Windows feature that tapping the ALT key invokes the Windows menu. The reason is that the ALT serves as META in Emacs. When using Emacs, users often press the META key temporarily and then change their minds; if this has the effect of bringing up the Windows menu, it alters the meaning of subsequent commands. Many users find this frustrating.

You can re-enable Windows's default handling of tapping the ALT key by setting `w32-pass-alt-to-system` to a non-`nil` value.

## I.7 Mouse Usage on MS-Windows

This section describes the Windows-specific variables related to the mouse.

The variable `w32-mouse-button-tolerance` specifies the time interval, in milliseconds, for faking middle mouse button press on 2-button mice. If both mouse buttons are depressed within this time interval, Emacs generates a middle mouse button click event instead of a double click on one of the buttons.

If the variable `w32-pass-extra-mouse-buttons-to-system` is non-`nil`, Emacs passes the fourth and fifth mouse buttons to Windows.

The variable `w32-swap-mouse-buttons` controls which of the 3 mouse buttons generates the `mouse-2` events. When it is `nil` (the default), the middle button generates `mouse-2` and the right button generates `mouse-3` events. If this variable is non-`nil`, the roles of these two buttons are reversed.

## I.8 Subprocesses on Windows 9X/ME and Windows NT/2K/XP

Emacs compiled as a native Windows application (as opposed to the DOS version) includes full support for asynchronous subprocesses. In the Windows version, synchronous and asynchronous subprocesses work fine on both Windows 9X/ME and Windows NT/2K/XP as long as you run only 32-bit Windows applications. However, when you run a DOS application in a subprocess, you may encounter problems or be unable to run the application at all; and if you run two DOS applications at the same time in two subprocesses, you may have to reboot your system.

Since the standard command interpreter (and most command line utilities) on Windows 9X are DOS applications, these problems are significant when using that system. But there's nothing we can do about them; only Microsoft can fix them.

If you run just one DOS application subprocess, the subprocess should work as expected as long as it is "well-behaved" and does not perform direct screen access or other unusual actions. If you have a CPU monitor application, your machine will appear to be 100% busy even when the DOS application is idle, but this is only an artifact of the way CPU monitors measure processor load.

You must terminate the DOS application before you start any other DOS application in a different subprocess. Emacs is unable to interrupt or terminate a DOS subprocess. The only way you can terminate such a subprocess is by giving it a command that tells its program to exit.

If you attempt to run two DOS applications at the same time in separate subprocesses, the second one that is started will be suspended until the first one finishes, even if either or both of them are asynchronous.

If you can go to the first subprocess, and tell it to exit, the second subprocess should continue normally. However, if the second subprocess is synchronous, Emacs itself will be hung until the first subprocess finishes. If it will not finish without user input, then you have no choice but to reboot if you are running on Windows 9X. If you are running on Windows NT/2K/XP, you can use a process viewer application to kill the appropriate instance of NTVDM instead (this will terminate both DOS subprocesses).

If you have to reboot Windows 9X in this situation, do not use the `Shutdown` command on the `Start` menu; that usually hangs the system. Instead, type `CTL-ALT-DEL` and then choose `Shutdown`. That usually works, although it may take a few minutes to do its job.

The variable `w32-quote-process-args` controls how Emacs quotes the process arguments. Non-`nil` means quote with the `"` character. If the value is a character, Emacs uses that character to escape any quote characters that appear; otherwise it chooses a suitable escape character based on the type of the program.

## I.9 Printing and MS-Windows

Printing commands, such as `lpr-buffer` (see Section 31.5 [Printing], page 669) and `ps-print-buffer` (see Section 31.5.1 [PostScript], page 670) work in MS-DOS and MS-Windows by sending the output to one of the printer ports, if a Posix-style `lpr` program is unavailable. The same Emacs variables control printing on all systems, but in some cases they have different default values on MS-DOS and MS-Windows.

Emacs on MS Windows attempts to determine your default printer automatically (using the function `default-printer-name`). But in some rare cases this can fail, or you may wish to use a different printer from within Emacs. The rest of this section explains how to tell Emacs which printer to use.

If you want to use your local printer, then set the Lisp variable `lpr-command` to `""` (its default value on Windows) and `printer-name` to the name of the printer port—for example, `"PRN"`, the usual local printer port, or `"LPT2"`, or `"COM1"` for a serial printer. You can also set `printer-name` to a file name, in which case "printed" output is actually appended to that file. If you set `printer-name` to `"NUL"`, printed output is silently discarded (sent to the system null device).

You can also use a printer shared by another machine by setting `printer-name` to the UNC share name for that printer—for example, `"//joes_pc/hp4si"`. (It doesn't matter whether you use forward slashes or backslashes here.) To find out the names of shared printers, run the command '`net view`' from the command prompt to obtain a list of servers, and '`net view` *`server-name`*' to see the names of printers (and directories) shared by that server. Alternatively, click the '`Network Neighborhood`' icon on your desktop, and look for machines that share their printers via the network.

If the printer doesn't appear in the output of '`net view`', or if setting `printer-name` to the UNC share name doesn't produce a hardcopy on that printer, you can use the '`net use`' command to connect a local print port such as `"LPT2"` to the networked printer. For example, typing `net use LPT2: \\joes_pc\hp4si`[2] causes Windows to *capture* the `LPT2` port and redirect the printed material to the printer connected to the machine `joes_pc`. After this command, setting `printer-name` to `"LPT2"` should produce the hardcopy on the networked printer.

With some varieties of Windows network software, you can instruct Windows to capture a specific printer port such as `"LPT2"`, and redirect it to a networked printer via the `Control Panel->Printers` applet instead of '`net use`'.

If you set `printer-name` to a file name, it's best to use an absolute file name. Emacs changes the working directory according to the default directory of the current buffer, so if the file name in `printer-name` is relative, you will end up with several such files, each one in the directory of the buffer from which the printing was done.

[2] Note that the '`net use`' command requires the UNC share name to be typed with the Windows-style backslashes, while the value of `printer-name` can be set with either forward- or backslashes.

If the value of `printer-name` is correct, but printing does not produce the hardcopy on your printer, it is possible that your printer does not support printing plain text (some cheap printers omit this functionality). In that case, try the PostScript print commands, described below.

The commands `print-buffer` and `print-region` call the `pr` program, or use special switches to the `lpr` program, to produce headers on each printed page. MS-DOS and MS-Windows don't normally have these programs, so by default, the variable `lpr-headers-switches` is set so that the requests to print page headers are silently ignored. Thus, `print-buffer` and `print-region` produce the same output as `lpr-buffer` and `lpr-region`, respectively. If you do have a suitable `pr` program (for example, from GNU Coreutils), set `lpr-headers-switches` to `nil`; Emacs will then call `pr` to produce the page headers, and print the resulting output as specified by `printer-name`.

Finally, if you do have an `lpr` work-alike, you can set the variable `lpr-command` to `"lpr"`. Then Emacs will use `lpr` for printing, as on other systems. (If the name of the program isn't `lpr`, set `lpr-command` to the appropriate value.) The variable `lpr-switches` has its standard meaning when `lpr-command` is not `""`. If the variable `printer-name` has a string value, it is used as the value for the `-P` option to `lpr`, as on Unix.

A parallel set of variables, `ps-lpr-command`, `ps-lpr-switches`, and `ps-printer-name` (see Section 31.5.2 [PostScript Variables], page 671), defines how PostScript files should be printed. These variables are used in the same way as the corresponding variables described above for non-PostScript printing. Thus, the value of `ps-printer-name` is used as the name of the device (or file) to which PostScript output is sent, just as `printer-name` is used for non-PostScript printing. (There are two distinct sets of variables in case you have two printers attached to two different ports, and only one of them is a PostScript printer.)

The default value of the variable `ps-lpr-command` is `""`, which causes PostScript output to be sent to the printer port specified by `ps-printer-name`; but `ps-lpr-command` can also be set to the name of a program which will accept PostScript files. Thus, if you have a non-PostScript printer, you can set this variable to the name of a PostScript interpreter program (such as Ghostscript). Any switches that need to be passed to the interpreter program are specified using `ps-lpr-switches`. (If the value of `ps-printer-name` is a string, it will be added to the list of switches as the value for the `-P` option. This is probably only useful if you are using `lpr`, so when using an interpreter typically you would set `ps-printer-name` to something other than a string so it is ignored.)

For example, to use Ghostscript for printing on the system's default printer, put this in your '`.emacs`' file:

```
(setq ps-printer-name t)
(setq ps-lpr-command "D:/gs6.01/bin/gswin32c.exe")
(setq ps-lpr-switches '("-q" "-dNOPAUSE" "-dBATCH"
                        "-sDEVICE=mswinpr2"
                        "-sPAPERSIZE=a4"))
```

(This assumes that Ghostscript is installed in the '`D:/gs6.01`' directory.)

## I.10 Specifying Fonts on MS-Windows

Starting with Emacs 23, fonts are specified by their name, size and optional properties. The format for specifying fonts comes from the fontconfig library used in modern Free desktops:

```
[Family[-PointSize]][:Option1=Value1[:Option2=Value2[...]]]
```

The old XLFD based format is also supported for backwards compatibility.

Emacs 23 and later supports a number of font backends. Currently, the `gdi` and `uniscribe` backends are supported on Windows. The `gdi` font backend is available on all versions of Windows, and supports all fonts that are natively supported by Windows. The `uniscribe` font backend is available on Windows 2000 and later, and supports TrueType and OpenType fonts. Some languages requiring complex layout can only be properly supported by the Uniscribe backend. By default, both backends are enabled if supported, with `uniscribe` taking priority over `gdi`. To override that and use the GDI backend even if Uniscribe is available, invoke Emacs with the `-xrm Emacs.fontBackend:gdi` command-line argument, or add a `Emacs.fontBackend` resource with the value `gdi` in the Registry under either the '`HKEY_CURRENT_USER\SOFTWARE\GNU\Emacs`' or the '`HKEY_LOCAL_MACHINE\SOFTWARE\GNU\Emacs`' key (see Section F.1 [Resources], page 769).

Optional properties common to all font backends on MS-Windows are:

`weight` Specifies the weight of the font. Special values `light`, `medium`, `demibold`, `bold`, and `black` can be specified without `weight=` (e.g., `Courier New-12:bold`). Otherwise, the weight should be a numeric value between 100 and 900, or one of the named weights in `font-weight-table`. If unspecified, a regular font is assumed.

`slant` Specifies whether the font is italic. Special values `roman`, `italic` and `oblique` can be specified without `slant=` (e.g., `Courier New-12:italic`). Otherwise, the slant should be a numeric value, or one of the named slants in `font-slant-table`. On Windows, any slant above 150 is treated as italics, and anything below as roman.

`family` Specifies the font family, but normally this will be specified at the start of the font name.

`pixelsize`
Specifies the font size in pixels. This can be used instead of the point size specified after the family name.

`adstyle` Specifies additional style information for the font. On MS-Windows, the values `mono`, `sans`, `serif`, `script` and `decorative` are recognized. These are most useful as a fallback with the font family left unspecified.

`registry` Specifies the character set registry that the font is expected to cover. Most TrueType and OpenType fonts will be Unicode fonts that cover several national character sets, but you can narrow down the selection of fonts to those that support a particular character set by using a specific registry from `w32-charset-info-alist` here.

`spacing` Specifies how the font is spaced. The `p` spacing specifies a proportional font, and `m` or `c` specify a monospaced font.

`foundry` Not used on Windows, but for informational purposes and to prevent problems with code that expects it to be set, is set internally to `raster` for bitmapped fonts, `outline` for scalable fonts, or `unknown` if the type cannot be determined as one of those.

Options specific to `GDI` fonts:

`script` Specifies a Unicode subrange the font should support.

The following scripts are recognized on Windows: `latin`, `greek`, `coptic`, `cyrillic`, `armenian`, `hebrew`, `arabic`, `syriac`, `nko`, `thaana`, `devanagari`, `bengali`, `gurmukhi`, `gujarati`, `oriya`, `tamil`, `telugu`, `kannada`, `malayam`, `sinhala`, `thai`, `lao`, `tibetan`, `myanmar`, `georgian`, `hangul`, `ethiopic`, `cherokee`, `canadian-aboriginal`, `ogham`, `runic`, `khmer`, `mongolian`, `symbol`, `braille`, `han`, `ideographic-description`, `cjk-misc`, `kana`, `bopomofo`, `kanbun`, `yi`, `byzantine-musical-symbol`, `musical-symbol`, and `mathematical`.

`antialias`

Specifies the antialiasing method. The value `none` means no antialiasing, `standard` means use standard antialiasing, `subpixel` means use subpixel antialiasing (known as Cleartype on Windows), and `natural` means use subpixel antialiasing with adjusted spacing between letters. If unspecified, the font will use the system default antialiasing.

## I.11 Miscellaneous Windows-specific features

This section describes miscellaneous Windows-specific features.

The variable `w32-use-visible-system-caret` is a flag that determines whether to make the system caret visible. The default when no screen reader software is in use is `nil`, which means Emacs draws its own cursor to indicate the position of point. A non-`nil` value means Emacs will indicate point location with the system caret; this facilitates use of screen reader software, and is the default when such software is detected when running Emacs. When this variable is non-`nil`, other variables affecting the cursor display have no effect.

See Info file '`emacs`', node '`Windows Misc`', for information about additional Windows-specific variables in this category.

# The GNU Manifesto

The GNU Manifesto which appears below was written by Richard Stallman at the beginning of the GNU project, to ask for participation and support. For the first few years, it was updated in minor ways to account for developments, but now it seems best to leave it unchanged as most people have seen it.

Since that time, we have learned about certain common misunderstandings that different wording could help avoid. Footnotes added in 1993 help clarify these points.

For up-to-date information about available GNU software, please see our web site, http://www.gnu.org. For software tasks and other ways to contribute, see http://www.gnu.org/help.

## What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it.[1] Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have an Emacs text editor with Lisp for writing editor commands, a source level debugger, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. A new portable optimizing C compiler has compiled itself and may be released this year. An initial kernel exists but many more features are needed to emulate Unix. When the kernel and compiler are finished, it will be possible to distribute a GNU system suitable for program development. We will use TEX as our text formatter, but an nroff is being worked on. We will use the free, portable X window system as well. After this we will add a portable Common Lisp, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer file names, file version numbers, a crashproof file system, file name completion perhaps, terminal-independent display support, and perhaps eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

[1] The wording here was careless. The intention was that nobody would have to pay for *permission* to use the GNU system. But the words don't make this clear, and people often interpret them as saying that copies of GNU should always be distributed at little or no charge. That was never the intent; later on, the manifesto mentions the possibility of companies providing the service of distribution for a profit. Subsequently I have learned to distinguish carefully between "free" in the sense of freedom and "free" in the sense of price. Free software is software that users have the freedom to distribute and change. Some users may obtain copies at no charge, while others pay to obtain copies—and if the funds help support improving the software, so much the better. The important thing is that everyone who has a copy has the freedom to cooperate with others in using it.

GNU is aimed initially at machines in the 68000/16000 class with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

To avoid horrible confusion, please pronounce the 'G' in the word 'GNU' when it is the name of this project.

## Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but eventually they had gone too far: I could not remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

## Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

## How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

## Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which

is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

## How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

## Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.[2]

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

[2] This is another place I failed to distinguish carefully between the two different meanings of "free." The statement as it stands is not false—you can get copies of GNU software at no charge, from your friends or over the net. But it does suggest the wrong idea.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

## Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."
>
> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.[3]

We must distinguish between support in the form of real programming work and mere handholding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distribution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."
>
> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more

[3] Several such companies now exist.

microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?[4]

"My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefiting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.[5]

"Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

"Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

---

[4] The Free Software Foundation raises most of its funds from a distribution service, although it is a charity rather than a company. If *no one* chooses to obtain copies by ordering from the FSF, it will be unable to do its work. But this does not mean that proprietary restrictions are justified to force every user to pay. If a small fraction of all the users order copies from the FSF, that is sufficient to keep the FSF afloat. So we ask users to choose to support us in this way. Have you done your part?

[5] A group of computer companies recently pooled funds to support maintenance of the GNU C Compiler.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights[6] carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors' works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the

[6] In the 80s I had not yet realized how confusing it was to speak of "the issue" of "intellectual property." That term is obviously biased; more subtle is the fact that it lumps together various disparate laws which raise very different issues. Nowadays I urge people to reject the term "intellectual property" entirely, lest it lead others to suppose that those laws form one coherent issue. The way to be clear is to discuss patents, copyrights, and trademarks separately. See http://www.gnu.org/philosophy/not-ipr.xhtml for more explanation of how this term spreads confusion and bias.

fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies—such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referee we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

"Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

"Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware[7], asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
>
> But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing—often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
>
> The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.
>
> The consequences:
>
> - The computer-using community supports software development.
> - This community decides what level of support is needed.
> - Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

[7] Subsequently we have discovered the need to distinguish between "free software" and "freeware". The term "freeware" means software you are free to redistribute, but usually you are not free to study and change the source code, so most of it is not free software. See http://www.gnu.org/philosophy/words-to-avoid.html for more explanation.

# Glossary

Abbrev
: An abbrev is a text string that expands into a different text string when present in the buffer. For example, you might define a few letters as an abbrev for a long phrase that you want to insert frequently. See Chapter 26 [Abbrevs], page 581.

Aborting
: Aborting means getting out of a recursive edit (q.v.). The commands `C-]` and `M-x top-level` are used for this. See Section 34.1 [Quitting], page 717.

Alt
: Alt is the name of a modifier bit that a keyboard input character may have. To make a character Alt, type it while holding down the ALT key. Such characters are given names that start with `Alt-` (usually written `A-` for short). (Note that many terminals have a key labeled ALT that is really a META key.) See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Argument
: See [Glossary—Numeric Argument], page 814.

ASCII character
: An ASCII character is either an ASCII control character or an ASCII printing character. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

ASCII control character
: An ASCII control character is the Control version of an upper-case letter, or the Control version of one of the characters '`@[\]^_?`'.

ASCII printing character
: ASCII letters, digits, space, and the following punctuation characters: '`!@#$%^&*()_-+=|\~`{}[]:;"'<>,.?/`'.

Auto Fill Mode
: Auto Fill mode is a minor mode (q.v.) in which text that you insert is automatically broken into lines of a given maximum width. See Section 22.11 [Filling], page 473.

Auto Saving
: Auto saving is the practice of periodically saving the contents of an Emacs buffer in a specially-named file, so that the information will be preserved if the buffer is lost due to a system error or user error. See ⟨undefined⟩ [Auto Save], page ⟨undefined⟩.

Autoloading
: Emacs can automatically load Lisp libraries when a Lisp program requests a function from those libraries. This is called 'autoloading'. See Section 24.8 [Lisp Libraries], page 549.

Backtrace
: A backtrace is a trace of a series of function calls showing how a program arrived at a certain point. It is used mainly for finding and correcting bugs (q.v.). Emacs can display a backtrace when it signals an error or when you type `C-g` (see [Glossary—Quitting], page 815). See Section 34.3.4 [Checklist], page 725.

Backup File
: A backup file records the contents that a file had before the current editing session. Emacs makes backup files automatically to help you track down or

cancel changes you later regret making. See ⟨undefined⟩ [Backup], page ⟨undefined⟩.

Balancing Parentheses
: Emacs can balance parentheses (or other matching delimiters) either manually or automatically. You do manual balancing with the commands to move over parenthetical groupings (see Section 23.4.2 [Moving by Parens], page 522). Automatic balancing works by blinking or highlighting the delimiter that matches the one you just inserted, or inserting the matching delimiter for you (see Section 23.4.3 [Matching Parens], page 522).

Balanced Expressions
: A balanced expression is a syntactically recognizable expression, such as a symbol, number, string constant, block, or parenthesized expression in C. See Section 23.4.1 [Expressions], page 521.

Balloon Help
: See [Glossary—Tooltips], page 819.

Base Buffer
: A base buffer is a buffer whose text is shared by an indirect buffer (q.v.).

Bidirectional Text
: Some human languages, such as English, are written from left to right. Others, such as Arabic, are written from right to left. Emacs supports both of these forms, as well as any mixture of them—this is 'bidirectional text'. See Section 19.20 [Bidirectional Editing], page 394.

Bind
: To bind a key sequence means to give it a binding (q.v.). See Section 33.3.5 [Rebinding], page 705.

Binding
: A key sequence gets its meaning in Emacs by having a binding, which is a command (q.v.), a Lisp function that is run when you type that sequence. See ⟨undefined⟩ [Commands], page ⟨undefined⟩. Customization often involves rebinding a character to a different command function. The bindings of all key sequences are recorded in the keymaps (q.v.). See Section 33.3.1 [Keymaps], page 703.

Blank Lines
: Blank lines are lines that contain only whitespace. Emacs has several commands for operating on the blank lines in the buffer. See Section 4.7 [Blank Lines], page 63.

Bookmark
: Bookmarks are akin to registers (q.v.) in that they record positions in buffers to which you can return later. Unlike registers, bookmarks persist between Emacs sessions. See Section 10.7 [Bookmarks], page 109.

Border
: A border is a thin space along the edge of the frame, used just for spacing, not for displaying anything. An Emacs frame has an ordinary external border, outside of everything including the menu bar, plus an internal border that surrounds the text windows, their scroll bars and fringes, and separates them from the menu bar and tool bar. You can customize both borders with options

and resources (see Section E.9 [Borders X], page 766). Borders are not the same as fringes (q.v.).

Buffer
: The buffer is the basic editing unit; one buffer corresponds to one text being edited. You normally have several buffers, but at any time you are editing only one, the 'current buffer', though several can be visible when you are using multiple windows or frames (q.v.). Most buffers are visiting (q.v.) some file. See Chapter 16 [Buffers], page 272.

Buffer Selection History
: Emacs keeps a buffer selection history that records how recently each Emacs buffer has been selected. This is used for choosing a buffer to select. See Chapter 16 [Buffers], page 272.

Bug
: A bug is an incorrect or unreasonable behavior of a program, or inaccurate or confusing documentation. Emacs developers treat bug reports, both in Emacs code and its documentation, very seriously and ask you to report any bugs you find. See Section 34.3 [Bugs], page 722.

Button Down Event
: A button down event is the kind of input event (q.v.) generated right away when you press down on a mouse button. See Section 33.3.10 [Mouse Buttons], page 709.

By Default
: See [Glossary—Default], page 804.

Byte Compilation
: See [Glossary—Compilation], page 802.

`C-`
: `C-` in the name of a character is an abbreviation for Control. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

`C-M-`
: `C-M-` in the name of a character is an abbreviation for Control-Meta. If your terminal lacks a real META key, you type a Control-Meta character by typing ESC and then typing the corresponding Control character. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Case Conversion
: Case conversion means changing text from upper case to lower case or vice versa. See ⟨undefined⟩ [Case], page ⟨undefined⟩.

Character
: Characters form the contents of an Emacs buffer. Also, key sequences (q.v.) are usually made up of characters (though they may include other input events as well). See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Character Set
: Emacs supports a number of character sets, each of which represents a particular alphabet or script. See Chapter 19 [International], page 374.

Character Terminal
: See [Glossary—Text Terminal], page 819.

Click Event
: A click event is the kind of input event (q.v.) generated when you press a mouse button and release it without moving the mouse. See Section 33.3.10 [Mouse Buttons], page 709.

Client
: See [Glossary—Server], page 818.

Clipboard
: A clipboard is a buffer provided by the window system for transferring text between applications. On the X Window System, the clipboard is provided in addition to the primary selection (q.v.); on MS-Windows and Mac, the clipboard is used *instead* of the primary selection. See Section 9.3.1 [Clipboard], page 100.

Coding System
: A coding system is an encoding for representing text characters in a file or in a stream of information. Emacs has the ability to convert text to or from a variety of coding systems when reading or writing it. See Section 19.6 [Coding Systems], page 381.

Command
: A command is a Lisp function specially defined to be able to serve as a key binding in Emacs. When you type a key sequence (q.v.), its binding (q.v.) is looked up in the relevant keymaps (q.v.) to find the command to run. See ⟨undefined⟩ [Commands], page ⟨undefined⟩.

Command History
: See [Glossary—Minibuffer History], page 813.

Command Name
: A command name is the name of a Lisp symbol that is a command (see ⟨undefined⟩ [Commands], page ⟨undefined⟩). You can invoke any command by its name using `M-x` (see Chapter 6 [Running Commands by Name], page 78).

Comment
: A comment is text in a program which is intended only for humans reading the program, and which is specially marked so that it will be ignored when the program is loaded or compiled. Emacs offers special commands for creating, aligning and killing comments. See Section 23.5 [Comments], page 523.

Common Lisp
: Common Lisp is a dialect of Lisp (q.v.) much larger and more powerful than Emacs Lisp. Emacs provides a subset of Common Lisp in the CL package. See Section "Overview" in *Common Lisp Extensions*.

Compilation
: Compilation is the process of creating an executable program from source code. Emacs has commands for compiling files of Emacs Lisp code (see Section "Byte Compilation" in *the Emacs Lisp Reference Manual*) and programs in C and other languages (see Section 24.1 [Compilation], page 534).

Complete Key
: A complete key is a key sequence that fully specifies one action to be performed by Emacs. For example, `X` and `C-f` and `C-x m` are complete keys. Complete keys derive their meanings from being bound (q.v.) to commands (q.v.). Thus,

`X` is conventionally bound to a command to insert '`X`' in the buffer; `C-x m` is conventionally bound to a command to begin composing a mail message. See ⟨undefined⟩ [Keys], page ⟨undefined⟩.

Completion
: Completion is what Emacs does when it automatically expands an abbreviation for a name into the entire name. Completion is done for minibuffer (q.v.) arguments when the set of possible valid inputs is known; for example, on command names, buffer names, and file names. Completion usually occurs when TAB, SPC or RET is typed. See Section 5.4 [Completion], page 70.

Continuation Line
: When a line of text is longer than the width of the window, it normally (but see [Glossary—Truncation], page 820) takes up more than one screen line when displayed. We say that the text line is continued, and all screen lines used for it after the first are called continuation lines. See Section 4.8 [Continuation Lines], page 63. A related Emacs feature is 'filling' (q.v.).

Control Character
: A control character is a character that you type by holding down the CTRL key. Some control characters also have their own keys, so that you can type them without using CTRL. For example, RET, TAB, ESC and DEL are all control characters. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Copyleft
: A copyleft is a notice giving the public legal permission to redistribute and modify a program or other work of art, but requiring modified versions to carry similar permission. Copyright is normally used to keep users divided and helpless; with copyleft we turn that around to empower users and encourage them to cooperate.

  The particular form of copyleft used by the GNU project is called the GNU General Public License. See Appendix A [Copying], page 733.

CTRL
: The CTRL or "control" key is what you hold down in order to enter a control character (q.v.). See [Glossary—C-], page 801.

Current Buffer
: The current buffer in Emacs is the Emacs buffer on which most editing commands operate. You can select any Emacs buffer as the current one. See Chapter 16 [Buffers], page 272.

Current Line
: The current line is the line that point is on (see Section 1.1 [Point], page 6).

Current Paragraph
: The current paragraph is the paragraph that point is in. If point is between two paragraphs, the current paragraph is the one that follows point. See ⟨undefined⟩ [Paragraphs], page ⟨undefined⟩.

Current Defun
: The current defun is the defun (q.v.) that point is in. If point is between defuns, the current defun is the one that follows point. See Section 23.2 [Defuns], page 515.

Cursor
: The cursor is the rectangle on the screen which indicates the position (called point; q.v.) at which insertion and deletion takes place. The cursor is on or under the character that follows point. Often people speak of 'the cursor' when, strictly speaking, they mean 'point'. See Section 1.1 [Point], page 6.

Customization
: Customization is making minor changes in the way Emacs works, to reflect your preferences or needs. It is often done by setting variables (see Section 33.2 [Variables], page 694) or faces (see Section 33.1.5 [Face Customization], page 690), or by rebinding key sequences (see Section 33.3.1 [Keymaps], page 703).

Cut and Paste
: See [Glossary—Killing], page 811, and [Glossary—Yanking], page 821.

Daemon
: A daemon is a standard term for a system-level process that runs in the background. Daemons are often started when the system first starts up. When Emacs runs in daemon-mode, it runs in the background and does not open a display. You can then connect to it with the `emacsclient` program. See Section 31.4 [Emacs Server], page 665.

Default Argument
: The default for an argument is the value that will be assumed if you do not specify one. When the minibuffer is used to read an argument, the default argument is used if you just type RET. See Chapter 5 [Minibuffer], page 68.

Default
: A default is the value that is used for a certain purpose when you do not explicitly specify a value to use.

Default Directory
: When you specify a file name that does not start with '`/`' or '`~`', it is interpreted relative to the current buffer's default directory. (On MS systems, file names that start with a drive letter '`x:`' are treated as absolute, not relative.) See Section 5.2 [Minibuffer File], page 68.

Defun
: A defun is a major definition at the top level in a program. The name 'defun' comes from Lisp, where most such definitions use the construct `defun`. See Section 23.2 [Defuns], page 515.

DEL
: DEL is a character that runs the command to delete one character of text before the cursor. It is typically either the DELETE key or the BACKSPACE key, whichever one is easy to type. See Section 4.3 [Erasing], page 61.

Deletion
: Deletion means erasing text without copying it into the kill ring (q.v.). The alternative is killing (q.v.). See Chapter 9 [Killing], page 95.

Deletion of Files
: Deleting a file means erasing it from the file system. (Note that some systems use the concept of a "trash can", or "recycle bin", to allow you to "undelete" files.) See ⟨undefined⟩ [Miscellaneous File Operations], page ⟨undefined⟩.

Deletion of Messages
: Deleting a message (in Rmail, and other mail clients) means flagging it to be eliminated from your mail file. Until you expunge (q.v.) the Rmail file, you can

still undelete the messages you have deleted. See Section 30.4 [Rmail Deletion], page 634.

Deletion of Windows
: Deleting a window means eliminating it from the screen. Other windows expand to use up the space. The text that was in the window is not lost, and you can create a new window with the same dimensions as the old if you wish. See Chapter 17 [Windows], page 289.

Directory
: File directories are named collections in the file system, within which you can place individual files or subdirectories. They are sometimes referred to as "folders". See ⟨undefined⟩ [Directories], page ⟨undefined⟩.

Directory Local Variable
: A directory local variable is a local variable (q.v.) that applies to all the files within a certain directory. See Section 33.2.5 [Directory Variables], page 701.

Dired
: Dired is the Emacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See Chapter 27 [Dired], page 588.

Disabled Command
: A disabled command is one that you may not run without special confirmation. The usual reason for disabling a command is that it is confusing for beginning users. See Section 33.3.11 [Disabling], page 711.

Down Event
: Short for 'button down event' (q.v.).

Drag Event
: A drag event is the kind of input event (q.v.) generated when you press a mouse button, move the mouse, and then release the button. See Section 33.3.10 [Mouse Buttons], page 709.

Dribble File
: A dribble file is a file into which Emacs writes all the characters that you type on the keyboard. Dribble files can be used to make a record for debugging Emacs bugs. Emacs does not make a dribble file unless you tell it to. See Section 34.3 [Bugs], page 722.

Echo Area
: The echo area is the bottom line of the screen, used for echoing the arguments to commands, for asking questions, and showing brief messages (including error messages). The messages are stored in the buffer '`*Messages*`' so you can review them later. See Section 1.2 [Echo Area], page 7.

Echoing
: Echoing is acknowledging the receipt of input events by displaying them (in the echo area). Emacs never echoes single-character key sequences; longer key sequences echo only if you pause while typing them.

Electric
: We say that a character is electric if it is normally self-inserting (q.v.), but the current major mode (q.v.) redefines it to do something else as well. For example, some programming language major modes define particular delimiter characters to reindent the line, or insert one or more newlines in addition to self-insertion.

End Of Line
: End of line is a character or a sequence of characters that indicate the end of a text line. On GNU and Unix systems, this is a newline (q.v.), but other systems have other conventions. See Section 19.6 [Coding Systems], page 381. Emacs can recognize several end-of-line conventions in files and convert between them.

Environment Variable
: An environment variable is one of a collection of variables stored by the operating system, each one having a name and a value. Emacs can access environment variables set by its parent shell, and it can set variables in the environment it passes to programs it invokes. See Section E.4 [Environment], page 758.

EOL
: See [Glossary—End Of Line], page 805.

Error
: An error occurs when an Emacs command cannot execute in the current circumstances. When an error occurs, execution of the command stops (unless the command has been programmed to do otherwise) and Emacs reports the error by displaying an error message (q.v.).

Error Message
: An error message is output displayed by Emacs when you ask it to do something impossible (such as, killing text forward when point is at the end of the buffer), or when a command malfunctions in some way. Such messages appear in the echo area, accompanied by a beep.

ESC
: ESC is a character used as a prefix for typing Meta characters on keyboards lacking a META key. Unlike the META key (which, like the SHIFT key, is held down while another character is typed), you press the ESC key as you would press a letter key, and it applies to the next character you type.

Expression
: See [Glossary—Balanced Expression], page 800.

Expunging
: Expunging an Rmail, Gnus newsgroup, or Dired buffer is an operation that truly discards the messages or files you have previously flagged for deletion.

Face
: A face is a style of displaying characters. It specifies attributes such as font family and size, foreground and background colors, underline and strike-through, background stipple, etc. Emacs provides features to associate specific faces with portions of buffer text, in order to display that text as specified by the face attributes. See Section 11.12 [Faces], page 137.

File Local Variable
: A file local variable is a local variable (q.v.) specified in a given file. See Section 33.2.4 [File Variables], page 698, and [Glossary—Directory Local Variable], page 805.

File Locking
: Emacs uses file locking to notice when two different users start to edit one file at the same time. See ⟨undefined⟩ [Interlocking], page ⟨undefined⟩.

File Name
: A file name is a name that refers to a file. File names may be relative or absolute; the meaning of a relative file name depends on the current directory,

but an absolute file name refers to the same file regardless of which directory is current. On GNU and Unix systems, an absolute file name starts with a slash (the root directory) or with '`~/`' or '`~user/`' (a home directory). On MS-Windows/MS-DOS, an absolute file name can also start with a drive letter and a colon, e.g., '`d:`'.

Some people use the term "pathname" for file names, but we do not; we use the word "path" only in the term "search path" (q.v.).

File-Name Component
: A file-name component names a file directly within a particular directory. On GNU and Unix systems, a file name is a sequence of file-name components, separated by slashes. For example, '`foo/bar`' is a file name containing two components, '`foo`' and '`bar`'; it refers to the file named '`bar`' in the directory named '`foo`' in the current directory. MS-DOS/MS-Windows file names can also use backslashes to separate components, as in '`foo\bar`'.

Fill Prefix
: The fill prefix is a string that should be expected at the beginning of each line when filling is done. It is not regarded as part of the text to be filled. See Section 22.11 [Filling], page 473.

Filling
: Filling text means adjusting the position of line-breaks to shift text between consecutive lines, so that all the lines are approximately the same length. See Section 22.11 [Filling], page 473. Some other editors call this feature "line wrapping".

Font Lock
: Font Lock is a mode that highlights parts of buffer text in different faces, according to the syntax. Some other editors refer to this as "syntax highlighting". For example, all comments (q.v.) might be colored red. See ⟨undefined⟩ [Font Lock], page ⟨undefined⟩.

Fontset
: A fontset is a named collection of fonts. A fontset specification lists character sets and which font to use to display each of them. Fontsets make it easy to change several fonts at once by specifying the name of a fontset, rather than changing each font separately. See Section 19.14 [Fontsets], page 389.

Formfeed Character
: See [Glossary—Page], page 814.

Frame
: A frame is a rectangular cluster of Emacs windows. Emacs starts out with one frame, but you can create more. You can subdivide each frame into Emacs windows (q.v.). When you are using a window system (q.v.), more than one frame can be visible at the same time. See Chapter 18 [Frames], page 341. Some other editors use the term "window" for this, but in Emacs a window means something else.

Free Software
: Free software is software that gives you the freedom to share, study and modify it. Emacs is free software, part of the GNU project (q.v.), and distributed under a copyleft (q.v.) license called the GNU General Public License. See Appendix A [Copying], page 733.

Free Software Foundation
: The Free Software Foundation (FSF) is a charitable foundation dedicated to promoting the development of free software (q.v.). For more information, see the FSF website.

Fringe
: On a graphical display (q.v.), there's a narrow portion of the frame (q.v.) between the text area and the window's border. These "fringes" are used to display symbols that provide information about the buffer text (see Section 11.13 [Fringes], page 156). Emacs displays the fringe using a special face (q.v.) called `fringe`. See Section 11.12 [Faces], page 137.

FSF
: See [Glossary—Free Software Foundation], page 807.

FTP
: FTP is an acronym for File Transfer Protocol. This is one standard method for retrieving remote files (q.v.).

Function Key
: A function key is a key on the keyboard that sends input but does not correspond to any character. See Section 33.3.8 [Function Keys], page 707.

Global
: Global means "independent of the current environment; in effect throughout Emacs". It is the opposite of local (q.v.). Particular examples of the use of 'global' appear below.

Global Abbrev
: A global definition of an abbrev (q.v.) is effective in all major modes that do not have local (q.v.) definitions for the same abbrev. See Chapter 26 [Abbrevs], page 581.

Global Keymap
: The global keymap (q.v.) contains key bindings that are in effect everywhere, except when overridden by local key bindings in a major mode's local keymap (q.v.). See Section 33.3.1 [Keymaps], page 703.

Global Mark Ring
: The global mark ring records the series of buffers you have recently set a mark (q.v.) in. In many cases you can use this to backtrack through buffers you have been editing, or in which you have found tags (see [Glossary—Tags Table], page 819). See Section 8.5 [Global Mark Ring], page 93.

Global Substitution
: Global substitution means replacing each occurrence of one string by another string throughout a large amount of text. See Section 12.10 [Replace], page 211.

Global Variable
: The global value of a variable (q.v.) takes effect in all buffers that do not have their own local (q.v.) values for the variable. See Section 33.2 [Variables], page 694.

GNU
: GNU is a recursive acronym for GNU's Not Unix, and it refers to a Unix-compatible operating system which is free software (q.v.). See [Manifesto], page 791. GNU is normally used with Linux as the kernel since Linux works better than the GNU kernel. For more information, see the GNU website.

Graphic Character
: Graphic characters are those assigned pictorial images rather than just names. All the non-Meta (q.v.) characters except for the Control (q.v.) characters are graphic characters. These include letters, digits, punctuation, and spaces; they do not include RET or ESC. In Emacs, typing a graphic character inserts that character (in ordinary editing modes). See Section 4.1 [Inserting Text], page 58.

Graphical Display
: A graphical display is one that can display images and multiple fonts. Usually it also has a window system (q.v.).

Highlighting
: Highlighting text means displaying it with a different foreground and/or background color to make it stand out from the rest of the text in the buffer.

  Emacs uses highlighting in several ways. It highlights the region whenever it is active (see Chapter 8 [Mark], page 89). Incremental search also highlights matches (see Section 12.1 [Incremental Search], page 199). See [Glossary—Font Lock], page 807.

Hardcopy
: Hardcopy means printed output. Emacs has various commands for printing the contents of Emacs buffers. See Section 31.5 [Printing], page 669.

HELP
: HELP is the Emacs name for `C-h` or F1. You can type HELP at any time to ask what options you have, or to ask what a command does. See ⟨undefined⟩ [Help], page ⟨undefined⟩.

Help Echo
: Help echo is a short message displayed in the echo area (q.v.) when the mouse pointer is located on portions of display that require some explanations. Emacs displays help echo for menu items, parts of the mode line, tool-bar buttons, etc. On graphical displays, the messages can be displayed as tooltips (q.v.). See ⟨undefined⟩ [Tooltips], page ⟨undefined⟩.

Home Directory
: Your home directory contains your personal files. On a multi-user GNU or Unix system, each user has his or her own home directory. When you start a new login session, your home directory is the default directory in which to start. A standard shorthand for your home directory is '`~`'. Similarly, '`~user`' represents the home directory of some other user.

Hook
: A hook is a list of functions to be called on specific occasions, such as saving a buffer in a file, major mode activation, etc. By customizing the various hooks, you can modify Emacs's behavior without changing any of its code. See Section 33.2.2 [Hooks], page 696.

Hyper
: Hyper is the name of a modifier bit that a keyboard input character may have. To make a character Hyper, type it while holding down the HYPER key. Such characters are given names that start with `Hyper-` (usually written `H-` for short). See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Iff
: "Iff" means "if and only if". This terminology comes from mathematics. Try to avoid using this term in documentation, since many are unfamiliar with it and mistake it for a typo.

Inbox
: An inbox is a file in which mail is delivered by the operating system. Rmail transfers mail from inboxes to Rmail files in which the mail is then stored permanently or until explicitly deleted. See Section 30.5 [Rmail Inbox], page 635.

Incremental Search
: Emacs provides an incremental search facility, whereby Emacs begins searching for a string as soon as you type the first character. As you type more characters, it refines the search. See Section 12.1 [Incremental Search], page 199.

Indentation
: Indentation means blank space at the beginning of a line. Most programming languages have conventions for using indentation to illuminate the structure of the program, and Emacs has special commands to adjust indentation. See Section 22.17 [Indentation], page 483.

Indirect Buffer
: An indirect buffer is a buffer that shares the text of another buffer, called its base buffer (q.v.). See Section 16.11 [Indirect Buffers], page 286.

Info
: Info is the hypertext format used by the GNU project for writing documentation.

Input Event
: An input event represents, within Emacs, one action taken by the user on the terminal. Input events include typing characters, typing function keys, pressing or releasing mouse buttons, and switching between Emacs frames. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Input Method
: An input method is a system for entering non-ASCII text characters by typing sequences of ASCII characters (q.v.). See Section 19.4 [Input Methods], page 378.

Insertion
: Insertion means adding text into the buffer, either from the keyboard or from some other place in Emacs.

Interlocking
: See [Glossary—File Locking], page 806.

Isearch
: See [Glossary—Incremental Search], page 810.

Justification
: Justification means adding extra spaces within lines of text in order to adjust the position of the text edges. See ⟨undefined⟩ [Fill Commands], page ⟨undefined⟩.

Key Binding
: See [Glossary—Binding], page 800.

Keyboard Macro
: Keyboard macros are a way of defining new Emacs commands from sequences of existing ones, with no need to write a Lisp program. You can use a macro to record a sequence of commands, then play them back as many times as you like. See Chapter 14 [Keyboard Macros], page 222.

Keyboard Shortcut
: A keyboard shortcut is a key sequence (q.v.) that invokes a command. What some programs call "assigning a keyboard shortcut", Emacs calls "binding a key sequence". See [Glossary—Binding], page 800.

Key Sequence
: A key sequence (key, for short) is a sequence of input events (q.v.) that are meaningful as a single unit. If the key sequence is enough to specify one action, it is a complete key (q.v.); if it is not enough, it is a prefix key (q.v.). See ⟨undefined⟩ [Keys], page ⟨undefined⟩.

Keymap
: The keymap is the data structure that records the bindings (q.v.) of key sequences to the commands that they run. For example, the global keymap binds the character `C-n` to the command function `next-line`. See Section 33.3.1 [Keymaps], page 703.

Keyboard Translation Table
: The keyboard translation table is an array that translates the character codes that come from the terminal into the character codes that make up key sequences.

Kill Ring
: The kill ring is where all text you have killed (see [Glossary—Killing], page 811) recently is saved. You can reinsert any of the killed text still in the ring; this is called yanking (q.v.). See Section 22.8.3 [Yanking], page 465.

Killing
: Killing means erasing text and saving it on the kill ring so it can be yanked (q.v.) later. Some other systems call this "cutting". Most Emacs commands that erase text perform killing, as opposed to deletion (q.v.). See Chapter 9 [Killing], page 95.

Killing a Job
: Killing a job (such as, an invocation of Emacs) means making it cease to exist. Any data within it, if not saved in a file, is lost. See Section 3.2 [Exiting], page 57.

Language Environment
: Your choice of language environment specifies defaults for the input method (q.v.) and coding system (q.v.). See Section 19.3 [Language Environments], page 377. These defaults are relevant if you edit non-ASCII text (see Chapter 19 [International], page 374).

Line Wrapping
: See [Glossary—Filling], page 807.

Lisp
: Lisp is a programming language. Most of Emacs is written in a dialect of Lisp, called Emacs Lisp, which is extended with special features that make it especially suitable for text editing tasks.

List
: A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. In C mode and other non-Lisp modes, groupings surrounded by other kinds of matched delimiters appropriate to the language, such as braces, are also considered lists. Emacs has special

commands for many operations on lists. See Section 23.4.2 [Moving by Parens], page 522.

Local
: Local means "in effect only in a particular context"; the relevant kind of context is a particular function execution, a particular buffer, or a particular major mode. It is the opposite of 'global' (q.v.). Specific uses of 'local' in Emacs terminology appear below.

Local Abbrev
: A local abbrev definition is effective only if a particular major mode is selected. In that major mode, it overrides any global definition for the same abbrev. See Chapter 26 [Abbrevs], page 581.

Local Keymap
: A local keymap is used in a particular major mode; the key bindings (q.v.) in the current local keymap override global bindings of the same key sequences. See Section 33.3.1 [Keymaps], page 703.

Local Variable
: A local value of a variable (q.v.) applies to only one buffer. See Section 33.2.3 [Locals], page 697.

`M-`
: `M-` in the name of a character is an abbreviation for META, one of the modifier keys that can accompany any character. See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

`M-C-`
: `M-C-` in the name of a character is an abbreviation for Control-Meta; it means the same thing as '`C-M-`' (q.v.).

`M-x`
: `M-x` is the key sequence that is used to call an Emacs command by name. This is how you run commands that are not bound to key sequences. See Chapter 6 [Running Commands by Name], page 78.

Mail
: Mail means messages sent from one user to another through the computer system, to be read at the recipient's convenience. Emacs has commands for composing and sending mail, and for reading and editing the mail you have received. See Chapter 29 [Sending Mail], page 624. See Chapter 30 [Rmail], page 632, for one way to read mail with Emacs.

Mail Composition Method
: A mail composition method is a program runnable within Emacs for editing and sending a mail message. Emacs lets you select from several alternative mail composition methods. See Section 29.7 [Mail Methods], page 631.

Major Mode
: The Emacs major modes are a mutually exclusive set of options, each of which configures Emacs for editing a certain sort of text. Ideally, each programming language has its own major mode. See Section 20.2 [Major Modes], page 399.

Margin
: The space between the usable part of a window (including the fringe) and the window edge.

Mark
: The mark points to a position in the text. It specifies one end of the region (q.v.), point being the other end. Many commands operate on all the text

from point to the mark. Each buffer has its own mark. See Chapter 8 [Mark], page 89.

Mark Ring
: The mark ring is used to hold several recent previous locations of the mark, in case you want to move back to them. Each buffer has its own mark ring; in addition, there is a single global mark ring (q.v.). See Section 8.4 [Mark Ring], page 92.

Menu Bar
: The menu bar is a line at the top of an Emacs frame. It contains words you can click on with the mouse to bring up menus, or you can use a keyboard interface to navigate it. See ⟨undefined⟩ [Menu Bars], page ⟨undefined⟩.

Message
: See [Glossary—Mail], page 812.

Meta
: Meta is the name of a modifier bit which you can use in a command character. To enter a meta character, you hold down the META key while typing the character. We refer to such characters with names that start with `Meta-` (usually written `M-` for short). For example, `M-<` is typed by holding down META and at the same time typing `<` (which itself is done, on most terminals, by holding down SHIFT and typing `,`). See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

  On some terminals, the META key is actually labeled ALT or EDIT.

Meta Character
: A Meta character is one whose character code includes the Meta bit.

Minibuffer
: The minibuffer is the window that appears when necessary inside the echo area (q.v.), used for reading arguments to commands. See Chapter 5 [Minibuffer], page 68.

Minibuffer History
: The minibuffer history records the text you have specified in the past for minibuffer arguments, so you can conveniently use the same text again. See Section 5.5 [Minibuffer History], page 74.

Minor Mode
: A minor mode is an optional feature of Emacs, which can be switched on or off independently of all other features. Each minor mode has a command to turn it on or off. Some minor modes are global (q.v.), and some are local (q.v.). See Section 20.3 [Minor Modes], page 413.

Minor Mode Keymap
: A minor mode keymap is a keymap that belongs to a minor mode and is active when that mode is enabled. Minor mode keymaps take precedence over the buffer's local keymap, just as the local keymap takes precedence over the global keymap. See Section 33.3.1 [Keymaps], page 703.

Mode Line
: The mode line is the line at the bottom of each window (q.v.), giving status information on the buffer displayed in that window. See Section 1.3 [Mode Line], page 8.

Modified Buffer
: A buffer (q.v.) is modified if its text has been changed since the last time the buffer was saved (or since it was created, if it has never been saved). See ⟨undefined⟩ [Saving], page ⟨undefined⟩.

Moving Text
: Moving text means erasing it from one place and inserting it in another. The usual way to move text is by killing (q.v.) it and then yanking (q.v.) it. See Chapter 9 [Killing], page 95.

MULE
: MULE refers to the Emacs features for editing multilingual non-ASCII text using multibyte characters (q.v.). See Chapter 19 [International], page 374.

Multibyte Character
: A multibyte character is a character that takes up several bytes in a buffer. Emacs uses multibyte characters to represent non-ASCII text, since the number of non-ASCII characters is much more than 256. See Section 19.1 [International Chars], page 374.

Named Mark
: A named mark is a register (q.v.), in its role of recording a location in text so that you can move point to that location. See Section 22.21 [Registers], page 507.

Narrowing
: Narrowing means creating a restriction (q.v.) that limits editing in the current buffer to only a part of the text. Text outside that part is inaccessible for editing (or viewing) until the boundaries are widened again, but it is still there, and saving the file saves it all. See ⟨undefined⟩ [Narrowing], page ⟨undefined⟩.

Newline
: Control-J characters in the buffer terminate lines of text and are therefore also called newlines. See [Glossary—End Of Line], page 805.

`nil`
: `nil` is a value usually interpreted as a logical "false". Its opposite is `t`, interpreted as "true".

Numeric Argument
: A numeric argument is a number, specified before a command, to change the effect of the command. Often the numeric argument serves as a repeat count. See Section 4.10 [Arguments], page 65.

Overwrite Mode
: Overwrite mode is a minor mode. When it is enabled, ordinary text characters replace the existing text after point rather than pushing it to one side. See Section 20.3 [Minor Modes], page 413.

Package
: A package is a collection of Lisp code that you download and automatically install from within Emacs. Packages provide a convenient way to add new features. See ⟨undefined⟩ [Packages], page ⟨undefined⟩.

Page
: A page is a unit of text, delimited by formfeed characters (ASCII control-L, code 014) at the beginning of a line. Some Emacs commands are provided for moving over and operating on pages. See ⟨undefined⟩ [Pages], page ⟨undefined⟩.

Paragraph
: Paragraphs are the medium-size unit of human-language text. There are special Emacs commands for moving over and operating on paragraphs. See ⟨undefined⟩ [Paragraphs], page ⟨undefined⟩.

Parsing
: We say that certain Emacs commands parse words or expressions in the text being edited. Really, all they know how to do is find the other end of a word or expression.

Point
: Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between two characters, not at one character. The terminal's cursor (q.v.) indicates the location of point. See Section 1.1 [Point], page 6.

Prefix Argument
: See [Glossary—Numeric Argument], page 814.

Prefix Key
: A prefix key is a key sequence (q.v.) whose sole function is to introduce a set of longer key sequences. `C-x` is an example of prefix key; any two-character sequence starting with `C-x` is therefore a legitimate key sequence. See ⟨undefined⟩ [Keys], page ⟨undefined⟩.

Primary Selection
: The primary selection is one particular X selection (q.v.); it is the selection that most X applications use for transferring text to and from other applications.

  The Emacs kill commands set the primary selection and the yank command uses the primary selection when appropriate. See Chapter 9 [Killing], page 95.

Prompt
: A prompt is text used to ask you for input. Displaying a prompt is called prompting. Emacs prompts always appear in the echo area (q.v.). One kind of prompting happens when the minibuffer is used to read an argument (see Chapter 5 [Minibuffer], page 68); the echoing that happens when you pause in the middle of typing a multi-character key sequence is also a kind of prompting (see Section 1.2 [Echo Area], page 7).

Query-Replace
: Query-replace is an interactive string replacement feature provided by Emacs. See Section 12.10.4 [Query Replace], page 213.

Quitting
: Quitting means canceling a partially typed command or a running command, using `C-g` (or `C-BREAK` on MS-DOS). See Section 34.1 [Quitting], page 717.

Quoting
: Quoting means depriving a character of its usual special significance. The most common kind of quoting in Emacs is with `C-q`. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an Emacs command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as DEL, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See Section 4.1 [Inserting Text], page 58.

Quoting File Names
: Quoting a file name turns off the special significance of constructs such as '`$`', '`~`' and '`:`'. See ⟨undefined⟩ [Quoted File Names], page ⟨undefined⟩.

Read-Only Buffer
: A read-only buffer is one whose text you are not allowed to change. Normally Emacs makes buffers read-only when they contain text which has a special significance to Emacs; for example, Dired buffers. Visiting a file that is write-protected also makes a read-only buffer. See Chapter 16 [Buffers], page 272.

Rectangle
: A rectangle consists of the text in a given range of columns on a given range of lines. Normally you specify a rectangle by putting point at one corner and putting the mark at the diagonally opposite corner. See Section 9.5 [Rectangles], page 103.

Recursive Editing Level
: A recursive editing level is a state in which part of the execution of a command involves asking you to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets (‘[’ and ‘]’). See Section 31.9 [Recursive Edit], page 675.

Redisplay
: Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See Chapter 1 [Screen], page 6.

Regexp
: See [Glossary—Regular Expression], page 816.

Region
: The region is the text between point (q.v.) and the mark (q.v.). Many commands operate on the text of the region. See Chapter 8 [Mark], page 89.

Register
: Registers are named slots in which text, buffer positions, or rectangles can be saved for later use. See Section 22.21 [Registers], page 507. A related Emacs feature is ‘bookmarks’ (q.v.).

Regular Expression
: A regular expression is a pattern that can match various text strings; for example, ‘`a[0-9]+`’ matches ‘`a`’ followed by one or more digits. See Section 12.6 [Regexps], page 206.

Remote File
: A remote file is a file that is stored on a system other than your own. Emacs can access files on other computers provided that they are connected to the same network as your machine, and (obviously) that you have a supported method to gain access to those files. See ⟨undefined⟩ [Remote Files], page ⟨undefined⟩.

Repeat Count
: See [Glossary—Numeric Argument], page 814.

Replacement
: See [Glossary—Global Substitution], page 808.

Restriction
: A buffer’s restriction is the amount of text, at the beginning or the end of the buffer, that is temporarily inaccessible. Giving a buffer a nonzero amount of restriction is called narrowing (q.v.); removing a restriction is called widening (q.v.). See ⟨undefined⟩ [Narrowing], page ⟨undefined⟩.

RET
: RET is a character that in Emacs runs the command to insert a newline into the text. It is also used to terminate most arguments read in the minibuffer (q.v.). See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Reverting
: Reverting means returning to the original state. Emacs lets you revert a buffer by re-reading its file from disk. See ⟨undefined⟩ [Reverting], page ⟨undefined⟩.

Saving
: Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. This is the way text in files actually gets changed by your Emacs editing. See ⟨undefined⟩ [Saving], page ⟨undefined⟩.

Scroll Bar
: A scroll bar is a tall thin hollow box that appears at the side of a window. You can use mouse commands in the scroll bar to scroll the window. The scroll bar feature is supported only under windowing systems. See Section 11.14 [Scroll Bars], page 162.

Scrolling
: Scrolling means shifting the text in the Emacs window so as to see a different part of the buffer. See ⟨undefined⟩ [Scrolling], page ⟨undefined⟩.

Searching
: Searching means moving point to the next occurrence of a specified string or the next match for a specified regular expression. See Chapter 12 [Search], page 199.

Search Path
: A search path is a list of directory names, to be used for searching for files for certain purposes. For example, the variable `load-path` holds a search path for finding Lisp library files. See Section 24.8 [Lisp Libraries], page 549.

Secondary Selection
: The secondary selection is one particular X selection (q.v.); some X applications can use it for transferring text to and from other applications. Emacs has special mouse commands for transferring text using the secondary selection. See Section 9.3.3 [Secondary Selection], page 101.

Selected Frame
: The selected frame is the one your input currently operates on. See Chapter 18 [Frames], page 341.

Selected Window
: The selected window is the one your input currently operates on. See ⟨undefined⟩ [Basic Window], page ⟨undefined⟩.

Selecting a Buffer
: Selecting a buffer means making it the current (q.v.) buffer. See ⟨undefined⟩ [Select Buffer], page ⟨undefined⟩.

Selection
: Windowing systems allow an application program to specify selections whose values are text. A program can also read the selections that other programs have set up. This is the principal way of transferring text between window applications. Emacs has commands to work with the primary (q.v.) selection and the secondary (q.v.) selection, and also with the clipboard (q.v.).

Self-Documentation
: Self-documentation is the feature of Emacs that can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask

for self-documentation with the help character, `C-h`. See ⟨undefined⟩ [Help], page ⟨undefined⟩.

Self-Inserting Character
: A character is self-inserting if typing that character inserts that character in the buffer. Ordinary printing and whitespace characters are self-inserting in Emacs, except in certain special major modes.

Sentences
: Emacs has commands for moving by or killing by sentences. See ⟨undefined⟩ [Sentences], page ⟨undefined⟩.

Server
: Within Emacs, you can start a 'server' process, which listens for connections from 'clients'. This offers a faster alternative to starting several Emacs instances. See Section 31.4 [Emacs Server], page 665, and [Glossary—Daemon], page 804.

Sexp
: A sexp (short for "s-expression") is the basic syntactic unit of Lisp in its textual form: either a list, or Lisp atom. Sexps are also the balanced expressions (q.v.) of the Lisp language; this is why the commands for editing balanced expressions have 'sexp' in their name. See Section 23.4.1 [Expressions], page 521.

Simultaneous Editing
: Simultaneous editing means two users modifying the same file at once. Simultaneous editing, if not detected, can cause one user to lose his or her work. Emacs detects all cases of simultaneous editing, and warns one of the users to investigate. See ⟨undefined⟩ [Simultaneous Editing], page ⟨undefined⟩.

SPC
: SPC is the space character, which you enter by pressing the space bar.

Speedbar
: The speedbar is a special tall frame that provides fast access to Emacs buffers, functions within those buffers, Info nodes, and other interesting parts of text within Emacs. See ⟨undefined⟩ [Speedbar], page ⟨undefined⟩.

Spell Checking
: Spell checking means checking correctness of the written form of each one of the words in a text. Emacs can use various external spelling-checker programs to check the spelling of parts of a buffer via a convenient user interface. See Section 13.4 [Spelling], page 219.

String
: A string is a kind of Lisp data object that contains a sequence of characters. Many Emacs variables are intended to have strings as values. The Lisp syntax for a string consists of the characters in the string with a '`"`' before and another '`"`' after. A '`"`' that is part of the string must be written as '`\"`' and a '`\`' that is part of the string must be written as '`\\`'. All other characters, including newline, can be included just by writing them inside the string; however, backslash sequences as in C, such as '`\n`' for newline or '`\241`' using an octal character code, are allowed as well.

String Substitution
: See [Glossary—Global Substitution], page 808.

Syntax Highlighting
: See [Glossary—Font Lock], page 807.

Syntax Table
: The syntax table tells Emacs which characters are part of a word, which characters balance each other like parentheses, etc. See Section "Syntax Tables" in *The Emacs Lisp Reference Manual*.

Super
: Super is the name of a modifier bit that a keyboard input character may have. To make a character Super, type it while holding down the SUPER key. Such characters are given names that start with `Super-` (usually written `s-` for short). See ⟨undefined⟩ [User Input], page ⟨undefined⟩.

Suspending
: Suspending Emacs means stopping it temporarily and returning control to its parent process, which is usually a shell. Unlike killing a job (q.v.), you can later resume the suspended Emacs job without losing your buffers, unsaved edits, undo history, etc. See Section 3.2 [Exiting], page 57.

TAB
: TAB is the tab character. In Emacs it is typically used for indentation or completion.

Tags Table
: A tags table is a file that serves as an index to the function definitions in one or more other files. See Section 25.3 [Tags], page 571.

Termscript File
: A termscript file contains a record of all characters sent by Emacs to the terminal. It is used for tracking down bugs in Emacs redisplay. Emacs does not make a termscript file unless you tell it to. See Section 34.3 [Bugs], page 722.

Text
: 'Text' has two meanings (see Chapter 22 [Text], page 454):

  - Data consisting of a sequence of characters, as opposed to binary numbers, executable programs, and the like. The basic contents of an Emacs buffer (aside from the text properties, q.v.) are always text in this sense.
  - Data consisting of written human language (as opposed to programs), or following the stylistic conventions of human language.

Text Terminal
: A text terminal, or character terminal, is a display that is limited to displaying text in character units. Such a terminal cannot control individual pixels it displays. Emacs supports a subset of display features on text terminals.

Text Properties
: Text properties are annotations recorded for particular characters in the buffer. Images in the buffer are recorded as text properties; they also specify formatting information. See ⟨undefined⟩ [Editing Format Info], page ⟨undefined⟩.

Theme
: A theme is a set of customizations (q.v.) that give Emacs a particular appearance or behavior. For example, you might use a theme for your favorite set of faces (q.v.).

Tool Bar
: The tool bar is a line (sometimes multiple lines) of icons at the top of an Emacs frame. Clicking on one of these icons executes a command. You can think of this as a graphical relative of the menu bar (q.v.). See ⟨undefined⟩ [Tool Bars], page ⟨undefined⟩.

Tooltips Tooltips are small windows displaying a help echo (q.v.) text, which explains parts of the display, lists useful options available via mouse clicks, etc. See ⟨undefined⟩ [Tooltips], page ⟨undefined⟩.

Top Level Top level is the normal state of Emacs, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level (q.v.) or the minibuffer (q.v.), and not in the middle of a command. You can get back to top level by aborting (q.v.) and quitting (q.v.). See Section 34.1 [Quitting], page 717.

Transposition
Transposing two units of text means putting each one into the place formerly occupied by the other. There are Emacs commands to transpose two adjacent characters, words, balanced expressions (q.v.) or lines (see Section 13.2 [Transpose], page 218).

Trash Can See [Glossary—Deletion of Files], page 804.

Truncation
Truncating text lines in the display means leaving out any text on a line that does not fit within the right margin of the window displaying it. See Section 4.8 [Continuation Lines], page 63, and [Glossary—Continuation Line], page 803.

TTY See [Glossary—Text Terminal], page 819.

Undoing Undoing means making your previous editing go in reverse, bringing back the text that existed earlier in the editing session. See Section 22.9 [Undo], page 469.

Unix Unix is a class of multi-user computer operating systems with a long history. There are several implementations today. The GNU project (q.v.) aims to develop a complete Unix-like operating system that is free software (q.v.).

User Option
A user option is a face (q.v.) or a variable (q.v.) that exists so that you can customize Emacs by setting it to a new value. See Section 33.1 [Easy Customization], page 686.

Variable A variable is an object in Lisp that can store an arbitrary value. Emacs uses some variables for internal purposes, and has others (known as 'user options'; q.v.) just so that you can set their values to control the behavior of Emacs. The variables used in Emacs that you are likely to be interested in are listed in the Variables Index in this manual (see [Variable Index], page 845). See Section 33.2 [Variables], page 694, for information on variables.

Version Control
Version control systems keep track of multiple versions of a source file. They provide a more powerful alternative to keeping backup files (q.v.). See Section 25.1 [Version Control], page 554.

Visiting Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See ⟨undefined⟩ [Visiting], page ⟨undefined⟩.

Whitespace
Whitespace is any run of consecutive formatting characters (space, tab, newline, and backspace).

Widening
: Widening is removing any restriction (q.v.) on the current buffer; it is the opposite of narrowing (q.v.). See ⟨undefined⟩ [Narrowing], page ⟨undefined⟩.

Window
: Emacs divides a frame (q.v.) into one or more windows, each of which can display the contents of one buffer (q.v.) at any time. See Chapter 1 [Screen], page 6, for basic information on how Emacs uses the screen. See Chapter 17 [Windows], page 289, for commands to control the use of windows. Some other editors use the term "window" for what we call a 'frame' (q.v.) in Emacs.

Window System
: A window system is software that operates on a graphical display (q.v.), to subdivide the screen so that multiple applications can have their] own windows at the same time. All modern operating systems include a window system.

Word Abbrev
: See [Glossary—Abbrev], page 799.

Word Search
: Word search is searching for a sequence of words, considering the punctuation between them as insignificant. See Section 12.3 [Word Search], page 203.

Yanking
: Yanking means reinserting text previously killed (q.v.). It can be used to undo a mistaken kill, or for copying or moving text. Some other systems call this "pasting". See Section 22.8.3 [Yanking], page 465.

# Key (Character) Index


## !

! (Dired) ..... 595

## #

# (Dired) ..... 590

## $

$ (Dired) ..... 599

## %

% & (Dired) ..... 590
% C (Dired) ..... 597
% d (Dired) ..... 591
% g (Dired) ..... 593
% H (Dired) ..... 597
% l (Dired) ..... 597
% m (Dired) ..... 593
% R (Dired) ..... 597
% S (Dired) ..... 597
% u (Dired) ..... 597

## *

* ! (Dired) ..... 592
* % (Dired) ..... 593
* * (Dired) ..... 592
* / (Dired) ..... 592
* ? (Dired) ..... 592
* @ (Dired) ..... 592
* c (Dired) ..... 592
* C-n (Dired) ..... 592
* C-p (Dired) ..... 592
* DEL (Dired) ..... 592
* m (Dired) ..... 591
* s (Dired) ..... 592
* t (Dired) ..... 592
* u (Dired) ..... 592

## +

+ (Dired) ..... 602
+ (DocView mode) ..... 654

## -

- (DocView mode) ..... 654

## .

. (Calendar mode) ..... 605
. (Dired) ..... 590
. (Rmail) ..... 633

## /

/ (Rmail) ..... 633

## :

:d (Dired) ..... 595
:e (Dired) ..... 595
:s (Dired) ..... 595
:v (Dired) ..... 595

## <

< (Calendar mode) ..... 606
< (Dired) ..... 598
< (Rmail) ..... 634

## =

= (Dired) ..... 597

## >

> (Calendar mode) ..... 606
> (Dired) ..... 599
> (Rmail) ..... 634

## ?

? (completion) ..... 71

## ^

^ (Dired) ..... 591

## ~

~ (Dired) ..... 590

## A

a (Calendar mode) ..... 609
A (Dired) ..... 595
a (Rmail) ..... 639
A k (Gnus Group mode) ..... 652
A s (Gnus Group mode) ..... 652
A u (Gnus Group mode) ..... 652
A z (Gnus Group mode) ..... 652
action (button property) ..... 180
after-string (overlay property) ..... 134

## B


B (Dired) ... 595
b (Rmail summary) ... 644
b (Rmail) ... 632
before-string (overlay property) ... 134
button (button property) ... 180


## C


C (Dired) ... 594
c (Rmail) ... 641
C-/ ... 217
C-@ ... 89
C-] ... 718
C-_ ... 217
C-_ (Dired) ... 593
C-\ ... 380
C-a ... 59
C-a (Calendar mode) ... 605
C-b ... 59
C-b (Calendar mode) ... 604
C-c , j ... 529
C-c , J ... 529
C-c , l ... 529
C-c , SPC ... 529
C-c . (C mode) ... 520
C-c . (Shell mode) ... 660
C-c < (GUD) ... 542
C-c > (GUD) ... 542
C-c @ C-h ... 527
C-c @ C-l ... 527
C-c @ C-M-h ... 527
C-c @ C-M-s ... 527
C-c @ C-r ... 527
C-c @ C-s ... 527
C-c C-\ (C mode) ... 532
C-c C-\ (Shell mode) ... 658
C-c C-a (C mode) ... 531
C-c C-a (Log Edit mode) ... 561
C-c C-a (Message mode) ... 630
C-c C-a (Shell mode) ... 657
C-c C-b (Message mode) ... 629
C-c C-b (Shell mode) ... 658
C-c C-c (C mode) ... 524
C-c C-c (customization buffer) ... 689
C-c C-c (Edit Tab Stops) ... 452
C-c C-c (Message mode) ... 627
C-c C-c (Shell mode) ... 658
C-c C-d (C Mode) ... 531
C-c C-d (GUD) ... 542
C-c C-d (Log Edit mode) ... 560
C-c C-DEL (C Mode) ... 531
C-c C-DELETE (C Mode) ... 531
C-c C-e (C mode) ... 532
C-c C-e (Shell mode) ... 658
C-c C-f (GUD) ... 542
C-c C-f (Log Edit mode) ... 560
C-c C-f (Shell mode) ... 658
C-c C-f C-b (Message mode) ... 629
C-c C-f C-c (Message mode) ... 629
C-c C-f C-f (Message mode) ... 629
C-c C-f C-r (Message mode) ... 629
C-c C-f C-s (Message mode) ... 629
C-c C-f C-t (Message mode) ... 629
C-c C-f C-w (Message mode) ... 629
C-c C-i (GUD) ... 541
C-c C-j (Term mode) ... 664
C-c C-k (Term mode) ... 664
C-c C-l (C mode) ... 531
C-c C-l (Calendar mode) ... 606
C-c C-l (GUD) ... 541
C-c C-l (Shell mode) ... 660
C-c C-n (C mode) ... 530
C-c C-n (GUD) ... 541
C-c C-n (Rmail) ... 634
C-c C-n (Shell mode) ... 661
C-c C-o (Shell mode) ... 658
C-c C-p (C mode) ... 530
C-c C-p (GUD) ... 541
C-c C-p (Rmail) ... 634
C-c C-p (Shell mode) ... 661
C-c C-q (C mode) ... 519
C-c C-q (Message mode) ... 629
C-c C-q (Term mode) ... 664
C-c C-r (GUD) ... 542
C-c C-r (Shell mode) ... 658
C-c C-s (C mode) ... 533
C-c C-s (GUD) ... 541
C-c C-s (Message mode) ... 627
C-c C-s (Shell mode) ... 658
C-c C-t (GUD) ... 542
C-c C-u (C mode) ... 530
C-c C-u (GUD) ... 542
C-c C-u (Shell mode) ... 658
C-c C-w (Message mode) ... 631
C-c C-w (Shell mode) ... 658
C-c C-x (Shell mode) ... 660
C-c C-y (Message mode) ... 629
C-c C-z (Shell mode) ... 658
C-c DEL (C Mode) ... 531
C-c DELETE (C Mode) ... 531
C-c RET (Goto Address mode) ... 678
C-c RET (Shell mode) ... 661
C-d (Rmail) ... 634
C-d (Shell mode) ... 657
C-e ... 59
C-e (Calendar mode) ... 605
C-f ... 59
C-f (Calendar mode) ... 604
C-g ... 717
C-g (Incremental search) ... 201
C-h C ... 382
C-h C-\ ... 380
C-h h ... 375
C-h I ... 380
C-h L ... 378

C-h S ... 526
C-h t ... 58
C-j ... 451
C-j (indenting source code) ... 517
C-j (Lisp Interaction mode) ... 552
C-k ... 96
C-k (Gnus Group mode) ... 652
C-LEFT ... 60
C-M-% ... 213
C-M-. ... 578
C-M-@ ... 91, 521
C-M-\ ... 452
C-M-a ... 516
C-M-b ... 521
C-M-c ... 675
C-M-d ... 522
C-M-d (Dired) ... 598
C-M-e ... 516
C-M-f ... 521
C-M-f (Rmail) ... 643
C-M-h ... 516
C-M-h (C mode) ... 516
C-M-i ... 528
C-M-i (customization buffer) ... 688
C-M-j ... 525
C-M-k ... 521
C-M-l (Rmail) ... 642
C-M-l (Shell mode) ... 658
C-M-n ... 522
C-M-n (Dired) ... 598
C-M-n (Rmail) ... 639
C-M-o ... 451
C-M-p ... 522
C-M-p (Dired) ... 598
C-M-p (Rmail) ... 639
C-M-q ... 518
C-M-q (C mode) ... 519
C-M-r ... 205
C-M-r (Rmail) ... 642
C-M-s ... 205
C-M-s (Rmail) ... 643
C-M-SPC ... 521
C-M-t ... 521
C-M-t (Rmail) ... 643
C-M-u ... 522
C-M-u (Dired) ... 598
C-M-w ... 99
C-M-w (Incremental search) ... 202
C-M-x (Emacs Lisp mode) ... 551
C-M-x (Lisp mode) ... 552
C-M-x (Scheme mode) ... 552
C-M-y (Incremental search) ... 202
C-n ... 59
C-n (Calendar mode) ... 604
C-n (Dired) ... 589
C-o ... 63
C-o (Dired) ... 591
C-o (Occur mode) ... 215
C-o (Rmail) ... 637
C-p ... 59
C-p (Calendar mode) ... 604
C-p (Dired) ... 589
C-q ... 58
C-r ... 200
C-RIGHT ... 59
C-s ... 199
C-S-backspace ... 96
C-S-Mouse-3 (FFAP) ... 679
C-SPC ... 89
C-SPC C-SPC ... 92
C-SPC C-SPC ... 94
C-t ... 218
C-t d (Image-Dired) ... 601
C-u ... 66
C-u C-/ ... 217
C-u C-SPC ... 92
C-u C-x C-x ... 94
C-u C-x v = ... 562
C-u M-; ... 524
C-u TAB ... 518
C-v (Calendar mode) ... 606
C-w ... 97
C-w (Incremental search) ... 202
C-x # ... 666
C-x ( ... 223
C-x ) ... 223
C-x ; ... 525
C-x = ... 65, 375
C-x [ (Calendar mode) ... 604
C-x [ (DocView mode) ... 653
C-x ] (Calendar mode) ... 604
C-x ] (DocView mode) ... 653
C-x ‘ ... 536
C-x 4 ... 578
C-x 4 a ... 570
C-x 4 d ... 588
C-x 4 f (FFAP) ... 679
C-x 4 m ... 624
C-x 5 ... 578
C-x 5 d ... 588
C-x 5 f (FFAP) ... 679
C-x 5 m ... 624
C-x 8 ... 393
C-x 8 RET ... 58
C-x C-a (GUD) ... 541
C-x C-a C-j (GUD) ... 542
C-x C-a C-w (GUD) ... 547
C-x C-c ... 57
C-x C-c (customization buffer) ... 689
C-x C-e ... 551
C-x C-f (FFAP) ... 678
C-x C-k b ... 227
C-x C-k C-a ... 225
C-x C-k C-c ... 225
C-x C-k C-e ... 228
C-x C-k C-f ... 225

C-x C-k C-i ... 225
C-x C-k C-k ... 224
C-x C-k C-n ... 224
C-x C-k C-p ... 224
C-x C-k e ... 228
C-x C-k l ... 228
C-x C-k n ... 227
C-x C-k r ... 223
C-x C-k RET ... 228
C-x C-k SPC ... 228
C-x C-n ... 60
C-x C-o ... 63
C-x C-r (FFAP) ... 678
C-x C-s (Custom Themes buffer) ... 693
C-x C-SPC ... 93
C-x C-t ... 218
C-x C-v (FFAP) ... 678
C-x C-x ... 90
C-x C-z ... 552
C-x d ... 588
C-x d (FFAP) ... 678
C-x e ... 223
C-x ESC ESC ... 76
C-x h ... 91
C-x m ... 624
C-x q ... 226
C-x r + ... 107, 108
C-x r b ... 109
C-x r c ... 104
C-x r d ... 104
C-x r f ... 107
C-x r i ... 107
C-x r j ... 106
C-x r k ... 104
C-x r l ... 109
C-x r m ... 109
C-x r M-w ... 104
C-x r n ... 108
C-x r N ... 104
C-x r o ... 104
C-x r r ... 107
C-x r s ... 107
C-x r SPC ... 106
C-x r t ... 104
C-x r w ... 107
C-x r y ... 104
C-x RET ... 375
C-x RET c ... 386
C-x RET C-\ ... 380
C-x RET f ... 386
C-x RET F ... 388
C-x RET k ... 389
C-x RET p ... 387
C-x RET r ... 387
C-x RET t ... 389
C-x RET x ... 387
C-x RET X ... 387
C-x SPC ... 541
C-x TAB ... 452
C-x u ... 217
C-x v + ... 568
C-x v = ... 562
C-x v ~ ... 562
C-x v d ... 565
C-x v D ... 562
C-x v g ... 562
C-x v i ... 561
C-x v l ... 564
C-x v u ... 565
C-x v v ... 558
C-x z ... 66
C-y ... 98
C-y (Incremental search) ... 202
C-z ... 57
category (overlay property) ... 132
category (text property) ... 494
composition (text property) ... 499
cursor (text property) ... 497

## D

d (Calendar mode) ... 616
d (Dired) ... 589
D (Dired) ... 594
D (GDB Breakpoints buffer) ... 545
D (GDB speedbar) ... 547
d (GDB threads buffer) ... 546
d (Rmail) ... 634
DEL (Dired) ... 589
DEL (DocView mode) ... 653
DEL (Gnus Group mode) ... 652
DEL (Gnus Summary mode) ... 652
DEL (programming modes) ... 514
DEL (Rmail) ... 633
disabled ... 53
display (overlay property) ... 132
display (text property) ... 163
DOWN ... 59
DOWN (minibuffer history) ... 75

## E

e (Dired) ... 591
e (Rmail) ... 647
END ... 59
ESC ESC ESC ... 718
evaporate (overlay property) ... 134
exit ... 52

## F

f (Dired) ... 591
f (GDB threads buffer) ... 546
f (Rmail) ... 641
F10 ... 9
F10 (MS-Windows) ... 785

F3 ... 222
F4 ... 222
face (button property) ... 180
face (overlay property) ... 132
face (text property) ... 494
field (overlay property) ... 133
field (text property) ... 497
file-already-exists ... 249
file-locked ... 240
file-supersession ... 280
follow-link (button property) ... 180
font-lock-face (text property) ... 495
fontified (text property) ... 495
function-documentation ... 79

## G

g (Dired) ... 599
G (Dired) ... 594
g (Rmail) ... 637
g *char* (Calendar mode) ... 613
g d (Calendar mode) ... 605
g D (Calendar mode) ... 605
g m (Calendar mode) ... 614
g w (Calendar mode) ... 605

## H

h (Calendar mode) ... 609
H (Dired) ... 594
h (Rmail) ... 642
help-echo (overlay property) ... 132
help-echo (text property) ... 495
help-index (button property) ... 180
HOME ... 59

## I

i (Dired) ... 598
i (Rmail) ... 636
i a (Calendar mode) ... 619
i b (Calendar mode) ... 619
i c (Calendar mode) ... 619
i d (Calendar mode) ... 618
i m (Calendar mode) ... 618
i w (Calendar mode) ... 618
i y (Calendar mode) ... 618
insert-behind-hooks (overlay property) ... 133
insert-behind-hooks (text property) ... 498
insert-in-front-hooks (overlay property) ... 133
insert-in-front-hooks (text property) ... 498
intangible (overlay property) ... 133
intangible (text property) ... 496
invisible (overlay property) ... 133
invisible (text property) ... 496

## J

j (Dired) ... 589
j (Rmail) ... 634

## K

k (Dired) ... 600
k (Rmail) ... 639
keymap (button property) ... 180
keymap (overlay property) ... 134
keymap (text property) ... 495

## L

l (Dired) ... 599
L (Dired) ... 595
l (GDB threads buffer) ... 546
l (Gnus Group mode) ... 652
L (Gnus Group mode) ... 652
l (Rmail) ... 642
LEFT ... 59
line-height (text property) ... 136, 497
line-spacing (text property) ... 137, 497
local-map (overlay property) ... 134
local-map (text property) ... 496

## M

m (Calendar mode) ... 616
M (Calendar mode) ... 610
m (Dired) ... 591
M (Dired) ... 594
m (Rmail) ... 641
M-! ... 655
M-$ ... 220
M-$ (Dired) ... 599
M-% ... 213
M-% (Incremental search) ... 201
M-& ... 655
M-* ... 578
M-, ... 578
M-- ... 65
M-- M-c ... 219
M-- M-l ... 219
M-- M-u ... 219
M-. ... 577
M-: ... 551
M-; ... 524
M-< ... 60
M-< (Calendar mode) ... 605
M-< (DocView mode) ... 653
M-= ... 65
M-= (Calendar mode) ... 606
M-> ... 60
M-> (Calendar mode) ... 605
M-> (DocView mode) ... 653
M-? (Shell mode) ... 657

M-@ ... 91
M-^ ... 452
M-‘ ... 9
M-{ (Calendar mode) ... 604
M-{ (Dired) ... 592
M-} (Calendar mode) ... 604
M-} (Dired) ... 592
M-\ ... 96
M-| ... 656
M-1 ... 65
M-a (C mode) ... 530
M-a (Calendar mode) ... 605
M-b ... 60
M-DEL (Dired) ... 592
M-Drag-Mouse-1 ... 101
M-e (C mode) ... 530
M-e (Calendar mode) ... 605
M-e (Incremental search) ... 200
M-f ... 59
M-g c ... 60
M-g g ... 60
M-g M-g ... 60
M-g M-n ... 536
M-g n ... 536
M-g TAB ... 60
M-i ... 451
M-j ... 525
M-LEFT ... 60
M-m ... 451
M-m (Rmail) ... 641
M-Mouse-1 ... 101
M-Mouse-2 ... 101
M-Mouse-3 ... 101
M-n (Incremental search) ... 200
M-n (Log Edit mode) ... 561
M-n (Man mode) ... 526
M-n (minibuffer history) ... 75
M-n (Rmail) ... 633
M-n (Shell mode) ... 660
M-p (Incremental search) ... 200
M-p (Log Edit mode) ... 561
M-p (Man mode) ... 526
M-p (minibuffer history) ... 75
M-p (Rmail) ... 633
M-p (Shell mode) ... 660
M-q (C mode) ... 532
M-r ... 60
M-r (Log Edit mode) ... 561
M-r (minibuffer history) ... 75
M-r (Shell mode) ... 660
M-RIGHT ... 59
M-s (Gnus Summary mode) ... 653
M-s (Log Edit mode) ... 561
M-s (minibuffer history) ... 75
M-s (Rmail) ... 634
M-s _ ... 204
M-s a C-s (Dired) ... 602
M-s a M-C-s (Dired) ... 602
M-s C-e (Incremental search) ... 202
M-s f C-s (Dired) ... 589
M-s f M-C-s (Dired) ... 589
M-s o ... 216
M-s SPC (Incremental search) ... 201
M-s w ... 204
M-SPC ... 96
M-t ... 218
M-TAB ... 528
M-TAB (customization buffer) ... 688
M-TAB (Incremental search) ... 202
M-v (Calendar mode) ... 606
M-w ... 97
M-x ... 78
M-y ... 98
M-y (Incremental search) ... 202
M-z ... 97
mode-class (property) ... 402
modification-hooks (overlay property) ... 133
modification-hooks (text property) ... 498
Mouse-2 (GDB Breakpoints buffer) ... 545
mouse-action (button property) ... 180
mouse-face (button property) ... 180
mouse-face (overlay property) ... 132
mouse-face (text property) ... 495

## N

n (DocView mode) ... 653
n (Gnus Group mode) ... 652
n (Gnus Summary mode) ... 652
n (Rmail) ... 633

## O

o (Calendar mode) ... 605
o (Dired) ... 591
O (Dired) ... 594
o (Occur mode) ... 215
o (Rmail) ... 637
operations (property) ... 265

## P

p (Calendar mode) ... 612
P (Dired) ... 595
p (DocView mode) ... 653
p (Gnus Group mode) ... 652
p (Gnus Summary mode) ... 652
p (Rmail) ... 633
p d (Calendar mode) ... 606
point-entered (text property) ... 499
point-left (text property) ... 499
pointer (text property) ... 497
priority (overlay property) ... 132

## Q


q (Calendar mode) ... 607
q (Dired) ... 588
Q (Dired) ... 595
q (Gnus Group mode) ... 652
q (Gnus Summary mode) ... 653
q (Rmail summary) ... 644
Q (Rmail summary) ... 644
q (Rmail) ... 632


## R


R (Dired) ... 594
r (GDB threads buffer) ... 546
r (Rmail) ... 640
read-only (text property) ... 496
RET ... 58
RET (completion in minibuffer) ... 72
RET (Dired) ... 591
RET (GDB Breakpoints buffer) ... 545
RET (GDB speedbar) ... 547
RET (Occur mode) ... 215
RET (Shell mode) ... 657
RIGHT ... 59


## S


s (Calendar mode) ... 616
S (Calendar mode) ... 609
s (Dired) ... 600
S (Dired) ... 594
s (Gnus Summary mode) ... 652
s (Rmail) ... 632
S-Mouse-2 ... 527
S-Mouse-3 (FFAP) ... 679
S-TAB (customization buffer) ... 687
safe-magic (property) ... 265
SPC (Calendar mode) ... 607
SPC (completion) ... 71
SPC (Dired) ... 589
SPC (DocView mode) ... 653
SPC (GDB Breakpoints buffer) ... 545
SPC (Gnus Group mode) ... 652
SPC (Gnus Summary mode) ... 652
SPC (Incremental search) ... 201
SPC (Rmail) ... 633


## T


t (Calendar mode) ... 607
t (Dired) ... 592
T (Dired) ... 595
t (Rmail) ... 646
TAB (completion) ... 70, 71
TAB (customization buffer) ... 687
TAB (GUD) ... 542
TAB (indentation) ... 451
TAB (Message mode) ... 629
TAB (programming modes) ... 517
TAB (Shell mode) ... 657
type (button property) ... 180


## U


u (Calendar mode) ... 609
u (Dired deletion) ... 589
u (Dired) ... 592
U (Dired) ... 592
u (Gnus Group mode) ... 652
u (Rmail) ... 635
UP ... 59
UP (minibuffer history) ... 75


## V


v (Dired) ... 591
v (Rmail) ... 646
variable-documentation ... 80


## W


w (Dired) ... 602
w (Rmail) ... 637
window (overlay property) ... 132


## X


x (Calendar mode) ... 609
x (Dired) ... 590
X (Dired) ... 595
x (Rmail) ... 635


## Z


Z (Dired) ... 595

# Command and Function Index

## 5

5x5 ..... 679

## A

abbrev-expansion ..... 584
abbrev-get ..... 586
abbrev-insert ..... 584
abbrev-prefix-mark ..... 584
abbrev-put ..... 586
abbrev-symbol ..... 584
abbrev-table-get ..... 587
abbrev-table-p ..... 581
abbrev-table-put ..... 587
abbreviate-file-name ..... 255
abort-recursive-edit ..... 53, 718
accept-change-group ..... 512
access-file ..... 241
activate-change-group ..... 512
add-change-log-entry-other-window ..... 570
add-dir-local-variable ..... 702
add-file-local-variable ..... 699
add-file-local-variable-prop-line ..... 698
add-hook ..... 398, 696
add-name-to-file ..... 248
add-text-properties ..... 491
add-to-invisibility-spec ..... 122
add-untranslated-filesystem ..... 783
adjust-window-trailing-edge ..... 296
after-find-file ..... 233
animate-birthday-present ..... 679
append-next-kill ..... 99
append-to-buffer ..... 102
append-to-file ..... 102, 237
append-to-register ..... 107
apply-macro-to-region-lines ..... 223
appt-activate ..... 620
appt-add ..... 620
appt-delete ..... 620
apropos ..... 85
ask-user-about-lock ..... 239
ask-user-about-supersession-threat ..... 280
async-shell-command ..... 655

## B

back-to-indentation ..... 451, 487
backward-button ..... 183
backward-char ..... 59
backward-delete-char-untabify ..... 461, 514
backward-list ..... 522
backward-sexp ..... 521
backward-to-indentation ..... 487
backward-up-list ..... 522
backward-word ..... 60
balance-windows ..... 297
balance-windows-area ..... 297
barf-if-buffer-read-only ..... 281
base64-decode-region ..... 509
base64-decode-string ..... 509
base64-encode-region ..... 509
base64-encode-string ..... 509
beep ..... 194
beginning-of-buffer ..... 60
beginning-of-defun ..... 516
bidi-string-mark-left-to-right ..... 197
bitmap-spec-p ..... 141
blackbox ..... 679
blink-matching-open ..... 188
bobp ..... 455
bolp ..... 455
bookmark-delete ..... 110
bookmark-insert ..... 110
bookmark-insert-location ..... 110
bookmark-jump ..... 109
bookmark-load ..... 110
bookmark-save ..... 109
bookmark-set ..... 109
bookmark-write ..... 110
browse-url ..... 677
browse-url-at-mouse ..... 677
browse-url-at-point ..... 677
bubbles ..... 679
buffer-base-buffer ..... 287
buffer-chars-modified-tick ..... 279
buffer-disable-undo ..... 472
buffer-enable-undo ..... 472
buffer-file-name ..... 276
buffer-list ..... 282
buffer-live-p ..... 286
buffer-modified-p ..... 278
buffer-modified-tick ..... 279
buffer-name ..... 275
buffer-string ..... 456
buffer-substring ..... 455
buffer-substring-no-properties ..... 456
buffer-swap-text ..... 287
bufferp ..... 272
bury-buffer ..... 283
button-activate ..... 182
button-at ..... 182
button-end ..... 182
button-get ..... 182
button-has-type-p ..... 182
button-label ..... 182
button-put ..... 182
button-start ..... 182
button-type ..... 182
button-type-get ..... 182

button-type-put ..... 182
button-type-subtype-p ..... 182

## C

c-backslash-region ..... 532
c-backward-conditional ..... 530
c-beginning-of-defun ..... 530
c-beginning-of-statement ..... 530
c-context-line-break ..... 532
c-electric-backspace ..... 514
c-end-of-defun ..... 530
c-end-of-statement ..... 530
c-fill-paragraph ..... 532
c-forward-conditional ..... 530
c-guess ..... 520
c-guess-install ..... 520
c-hungry-delete-backwards ..... 531
c-hungry-delete-forward ..... 531
c-indent-command ..... 517, 519
c-indent-defun ..... 519
c-indent-exp ..... 519
c-macro-expand ..... 532
c-mark-function ..... 516
c-set-style ..... 520
c-show-syntactic-information ..... 533
c-toggle-auto-newline ..... 531
c-toggle-electric-state ..... 531
c-toggle-hungry-state ..... 531
c-up-conditional ..... 530
calendar ..... 604
calendar-astro-goto-day-number ..... 613
calendar-astro-print-day-number ..... 612
calendar-backward-day ..... 604
calendar-backward-month ..... 604
calendar-backward-week ..... 604
calendar-bahai-goto-date ..... 613
calendar-bahai-print-date ..... 612
calendar-beginning-of-month ..... 605
calendar-beginning-of-week ..... 605
calendar-beginning-of-year ..... 605
calendar-chinese-goto-date ..... 613
calendar-chinese-print-date ..... 612
calendar-coptic-goto-date ..... 613
calendar-coptic-print-date ..... 612
calendar-count-days-region ..... 606
calendar-cursor-holidays ..... 609
calendar-end-of-month ..... 605
calendar-end-of-week ..... 605
calendar-end-of-year ..... 605
calendar-ethiopic-goto-date ..... 613
calendar-ethiopic-print-date ..... 612
calendar-forward-day ..... 604
calendar-forward-month ..... 604
calendar-forward-week ..... 604
calendar-forward-year ..... 604
calendar-french-goto-date ..... 613
calendar-french-print-date ..... 612
calendar-goto-date ..... 605
calendar-goto-day-of-year ..... 605
calendar-goto-today ..... 605
calendar-hebrew-goto-date ..... 613
calendar-hebrew-list-yahrzeits ..... 613
calendar-hebrew-print-date ..... 612
calendar-islamic-goto-date ..... 613
calendar-islamic-print-date ..... 612
calendar-iso-goto-date ..... 613
calendar-iso-goto-week ..... 605, 613
calendar-iso-print-date ..... 612
calendar-julian-goto-date ..... 613
calendar-julian-print-date ..... 612
calendar-list-holidays ..... 609
calendar-lunar-phases ..... 610
calendar-mark-holidays ..... 609
calendar-mayan-goto-long-count-date ..... 614
calendar-mayan-next-calendar-round-date ..... 614
calendar-mayan-next-haab-date ..... 614
calendar-mayan-next-tzolkin-date ..... 614
calendar-mayan-previous-haab-date ..... 614
calendar-mayan-previous-tzolkin-date ..... 614
calendar-mayan-print-date ..... 612
calendar-other-month ..... 605
calendar-persian-goto-date ..... 613
calendar-persian-print-date ..... 612
calendar-print-day-of-year ..... 606
calendar-print-other-dates ..... 612
calendar-redraw ..... 606
calendar-scroll-left ..... 606
calendar-scroll-left-three-months ..... 606
calendar-scroll-right ..... 606
calendar-scroll-right-three-months ..... 606
calendar-set-date-style ..... 617
calendar-sunrise-sunset ..... 609
calendar-unmark ..... 609
call-interactively ..... 18
called-interactively-p ..... 20
cancel-change-group ..... 512
capitalize-region ..... 488
capitalize-word ..... 488
change-log-merge ..... 570
change-log-mode ..... 570
char-after ..... 454
char-before ..... 454
char-displayable-p ..... 154
char-width ..... 135
check-parens ..... 520
choose-completion ..... 71
clear-abbrev-table ..... 581
clear-image-cache ..... 179
clear-rectangle ..... 104
clear-this-command-keys ..... 22
clear-visited-file-modtime ..... 279
clipboard-kill-region ..... 100
clipboard-kill-ring-save ..... 100
clipboard-yank ..... 100

clone-indirect-buffer ..... 287
collapse-delayed-warnings ..... 121
color-defined-p ..... 367
color-gray-p ..... 368
color-supported-p ..... 368
color-values ..... 368
combine-after-change-calls ..... 513
comint-bol-or-process-mark ..... 657
comint-continue-subjob ..... 659
comint-copy-old-input ..... 661
comint-delchar-or-maybe-eof ..... 657
comint-delete-output ..... 658
comint-dynamic-list-filename ..... 657
comint-dynamic-list-input-ring ..... 660
comint-get-next-from-history ..... 660
comint-history-isearch-backward-regexp .. 660
comint-input-previous-argument ..... 660
comint-interrupt-subjob ..... 658
comint-kill-input ..... 658
comint-magic-space ..... 662
comint-next-input ..... 660
comint-next-prompt ..... 661
comint-previous-input ..... 660
comint-previous-prompt ..... 661
comint-quit-subjob ..... 658
comint-run ..... 659
comint-send-input ..... 657
comint-show-maximum-output ..... 658
comint-show-output ..... 658
comint-stop-subjob ..... 658
comint-strip-ctrl-m ..... 659
comint-truncate-buffer ..... 659
comint-write-output ..... 658
command-execute ..... 18
commandp ..... 18
comment-dwim ..... 524
comment-indent-new-line ..... 525
comment-kill ..... 524
comment-region ..... 524
comment-set-column ..... 525
compare-buffer-substrings ..... 457
compare-window-configurations ..... 336
compilation-next-error ..... 536
compilation-next-file ..... 536
compilation-previous-error ..... 536
compilation-previous-file ..... 536
compile ..... 534
compile-goto-error ..... 535
completion-at-point ..... 528, 657
compose-mail ..... 624
compose-mail-other-frame ..... 624
compose-mail-other-window ..... 624
constrain-to-field ..... 505
convert-standard-filename ..... 260
coordinates-in-window-p ..... 334
copy-abbrev-table ..... 581
copy-dir-locals-to-file-locals ..... 699
copy-dir-locals-to-file-locals-prop-line ..... 698
copy-directory ..... 262
copy-file ..... 249
copy-file-locals-to-dir-locals ..... 702
copy-overlay ..... 129
copy-rectangle-as-kill ..... 104
copy-rectangle-to-register ..... 107
copy-region-as-kill ..... 465
copy-to-buffer ..... 102
copy-to-register ..... 107
count-words ..... 65
count-words-region ..... 65
cpp-highlight-buffer ..... 532
create-file-buffer ..... 233
create-fontset-from-fontset-spec .... 152, 391
create-image ..... 175
crisp-mode ..... 676
cua-mode ..... 105
current-bidi-paragraph-direction ..... 197
current-buffer ..... 272
current-column ..... 482
current-fill-column ..... 476
current-frame-configuration ..... 362
current-indentation ..... 483
current-justification ..... 474
current-kill ..... 467
current-left-margin ..... 476
current-message ..... 115
current-window-configuration ..... 335
current-word ..... 457
Custom-save ..... 689
Custom-set ..... 689
customize ..... 686
customize-apropos ..... 692
customize-browse ..... 687
customize-changed ..... 692
customize-create-theme ..... 693
customize-face ..... 692
customize-group ..... 692
customize-option ..... 692
customize-saved ..... 692
customize-themes ..... 692
customize-unsaved ..... 692
cwarn-mode ..... 533
cygwin-convert-file-name-from-windows ... 251
cygwin-convert-file-name-to-windows ..... 251

## D

dbx ..... 539
debug_print ..... 728
decipher ..... 679
default-file-modes ..... 250
default-value ..... 698
defface ..... 141
defimage ..... 175
define-abbrev ..... 582

define-abbrev-table ..... 582
define-button-type ..... 180
define-derived-mode ..... 405
define-fringe-bitmap ..... 160
define-generic-mode ..... 411
define-globalized-minor-mode ..... 418
define-key ..... 707
define-minor-mode ..... 416
define-obsolete-face-alias ..... 149
define-package ..... 683
defined-colors ..... 368
delay-mode-hooks ..... 408
delete-and-extract-region ..... 461
delete-backward-char ..... 95, 461
delete-blank-lines ..... 63, 463
delete-char ..... 95, 461
delete-dir-local-variable ..... 702
delete-directory ..... 262
delete-field ..... 505
delete-file ..... 249
delete-file-local-variable ..... 699
delete-file-local-variable-prop-line ..... 698
delete-frame ..... 357
delete-horizontal-space ..... 96, 462
delete-indentation ..... 452, 462
delete-other-windows ..... 300
delete-overlay ..... 129
delete-rectangle ..... 104
delete-region ..... 461
delete-selection-mode ..... 92
delete-terminal ..... 343
delete-to-left-margin ..... 476
delete-whitespace-rectangle ..... 104
delete-window ..... 300
delete-windows-on ..... 300
derived-mode-p ..... 407
describe-categories ..... 210
describe-character-set ..... 394
describe-coding-system ..... 382
describe-current-display-table ..... 191
describe-display-table ..... 191
describe-input-method ..... 380
describe-language-environment ..... 378
describe-mode ..... 405
describe-prefix-bindings ..... 86
describe-theme ..... 693
desktop-change-dir ..... 675
desktop-clear ..... 675
desktop-revert ..... 675
desktop-save ..... 675
destroy-fringe-bitmap ..... 161
diary ..... 616
diary-anniversary ..... 619
diary-block ..... 619
diary-cyclic ..... 619
diary-float ..... 619
diary-insert-anniversary-entry ..... 619
diary-insert-block-entry ..... 619
diary-insert-cyclic-entry ..... 619
diary-insert-entry ..... 618
diary-insert-monthly-entry ..... 618
diary-insert-weekly-entry ..... 618
diary-insert-yearly-entry ..... 618
diary-mail-entries ..... 616
diary-mark-entries ..... 616
diary-show-all-entries ..... 616
diary-view-entries ..... 616
digit-argument ..... 51, 65
ding ..... 194
dir-locals-set-class-variables ..... 702
dir-locals-set-directory-class ..... 702
directory-file-name ..... 254
directory-files ..... 260
directory-files-and-attributes ..... 261
dired ..... 588
dired-at-point ..... 678
dired-change-marks ..... 592
dired-clean-directory ..... 590
dired-compare-directories ..... 603
dired-copy-filename-as-kill ..... 602
dired-create-directory ..... 602
dired-diff ..... 597
dired-display-file ..... 591
dired-do-byte-compile ..... 595
dired-do-chgrp ..... 594
dired-do-chmod ..... 594
dired-do-chown ..... 594
dired-do-compress ..... 595
dired-do-copy ..... 594
dired-do-copy-regexp ..... 597
dired-do-delete ..... 594
dired-do-flagged-delete ..... 590
dired-do-hardlink ..... 594
dired-do-hardlink-regexp ..... 597
dired-do-isearch ..... 602
dired-do-isearch-regexp ..... 602
dired-do-kill-lines ..... 600
dired-do-load ..... 595
dired-do-print ..... 595
dired-do-query-replace-regexp ..... 595
dired-do-redisplay ..... 599
dired-do-rename ..... 594
dired-do-rename-regexp ..... 597
dired-do-search ..... 595
dired-do-shell-command ..... 595
dired-do-symlink ..... 594
dired-do-symlink-regexp ..... 597
dired-do-touch ..... 595
dired-downcase ..... 597
dired-find-file ..... 591
dired-find-file-other-window ..... 591
dired-flag-auto-save-files ..... 590
dired-flag-backup-files ..... 590
dired-flag-file-deletion ..... 589
dired-flag-files-regexp ..... 591
dired-flag-garbage-files ..... 590

dired-goto-file ... 589
dired-hide-all ... 599
dired-hide-subdir ... 599
dired-isearch-filenames ... 589
dired-isearch-filenames-regexp ... 589
dired-mark ... 591
dired-mark-directories ... 592
dired-mark-executables ... 592
dired-mark-files-containing-regexp ... 593
dired-mark-files-regexp ... 593
dired-mark-subdir-files ... 592
dired-mark-symlinks ... 592
dired-maybe-insert-subdir ... 598
dired-mouse-find-file-other-window ... 591
dired-next-dirline ... 599
dired-next-marked-file ... 592
dired-next-subdir ... 598
dired-other-frame ... 588
dired-other-window ... 588
dired-prev-dirline ... 598
dired-prev-marked-file ... 592
dired-prev-subdir ... 598
dired-sort-toggle-or-edit ... 600
dired-toggle-marks ... 592
dired-tree-down ... 598
dired-tree-up ... 598
dired-undo ... 593
dired-unmark ... 592
dired-unmark-all-files ... 592
dired-unmark-all-marks ... 592
dired-unmark-backward ... 592
dired-up-directory ... 591
dired-upcase ... 597
dired-view-file ... 591
dirs ... 662
dirtrack-mode ... 662
disable-command ... 53, 711
disable-theme ... 693
discard-input ... 46
display-backing-store ... 372
display-buffer ... 313
display-buffer-below-selected ... 316
display-buffer-in-previous-window ... 316
display-buffer-pop-up-frame ... 315
display-buffer-pop-up-window ... 315
display-buffer-reuse-window ... 314
display-buffer-same-window ... 314
display-buffer-use-some-window ... 316
display-color-cells ... 372
display-color-p ... 371
display-delayed-warnings ... 121
display-graphic-p ... 371
display-grayscale-p ... 371
display-images-p ... 371
display-message-or-buffer ... 115
display-mm-height ... 372
display-mm-width ... 372
display-mouse-p ... 371
display-pixel-height ... 371
display-pixel-width ... 372
display-planes ... 372
display-popup-menus-p ... 371
display-save-under ... 372
display-screens ... 371
display-selections-p ... 371
display-supports-face-attributes-p ... 371
display-table-slot ... 191
display-visual-class ... 372
display-warning ... 119
dissociated-press ... 679
doc-view-clear-cache ... 654
doc-view-enlarge ... 654
doc-view-first-page ... 653
doc-view-goto-page ... 653
doc-view-kill-proc ... 654
doc-view-kill-proc-and-buffer ... 654
doc-view-last-page ... 653
doc-view-mode ... 653
doc-view-next-page ... 653
doc-view-previous-page ... 653
doc-view-reset-slice ... 654
doc-view-scroll-down-or-previous-page ... 653
doc-view-scroll-up-or-next-page ... 653
doc-view-search ... 654
doc-view-search-backward ... 654
doc-view-set-slice ... 654
doc-view-set-slice-using-mouse ... 654
doc-view-show-tooltip ... 654
doc-view-shrink ... 654
doc-view-toggle-display ... 653
doctor ... 680
documentation ... 80
documentation-property ... 80
dotimes-with-progress-reporter ... 117
down-list ... 522
downcase-region ... 488
downcase-word ... 488
dunnet ... 679

## E

easy-mmode-define-minor-mode ... 417
edit-kbd-macro ... 228
edit-tab-stops ... 452
edt-emulation-off ... 676
edt-emulation-on ... 676
eldoc-mode ... 527
electric-indent-mode ... 453
electric-layout-mode ... 529
electric-pair-mode ... 523
emacs-lisp-mode ... 550
emacs-version ... 724
enable-command ... 53, 711
enable-theme ... 693
end-of-buffer ... 60
end-of-defun ... 516

eobp ... 455
eolp ... 455
epa-dired-do-decrypt ... 595
epa-dired-do-encrypt ... 595
epa-dired-do-sign ... 595
epa-dired-do-verify ... 595
erase-buffer ... 461
eval-buffer ... 551
eval-defun ... 551
eval-expression ... 551
eval-last-sexp ... 551
eval-print-last-sexp ... 552
eval-region ... 551
event-basic-type ... 34
event-click-count ... 29
event-convert-list ... 34
event-end ... 35
event-modifiers ... 33
event-start ... 35
eventp ... 23
ewoc-buffer ... 184
ewoc-collect ... 186
ewoc-create ... 184
ewoc-data ... 185
ewoc-delete ... 186
ewoc-enter-after ... 185
ewoc-enter-before ... 185
ewoc-enter-first ... 185
ewoc-enter-last ... 185
ewoc-filter ... 186
ewoc-get-hf ... 184
ewoc-goto-next ... 185
ewoc-goto-node ... 185
ewoc-goto-prev ... 185
ewoc-invalidate ... 186
ewoc-locate ... 185
ewoc-location ... 185
ewoc-map ... 186
ewoc-next ... 185
ewoc-nth ... 185
ewoc-prev ... 185
ewoc-refresh ... 185
ewoc-set-data ... 185
ewoc-set-hf ... 185
exchange-point-and-mark ... 90
executable-find ... 247
execute-extended-command ... 19, 78
execute-kbd-macro ... 54
exit-calendar ... 607
exit-recursive-edit ... 52, 675
expand-abbrev ... 584
expand-file-name ... 255

## F

face-all-attributes ... 144
face-attribute ... 143
face-attribute-relative-p ... 144
face-background ... 145
face-bold-p ... 146
face-differs-from-default-p ... 148
face-documentation ... 81, 148
face-equal ... 148
face-font ... 145
face-foreground ... 145
face-id ... 148
face-inverse-video-p ... 146
face-italic-p ... 146
face-list ... 148
face-remap-add-relative ... 147
face-remap-remove-relative ... 148
face-remap-reset-base ... 148
face-remap-set-base ... 148
face-stipple ... 145
face-underline-p ... 146
facep ... 138
ff-find-related-file ... 533
ffap ... 678
ffap-menu ... 678
ffap-mode ... 678
ffap-next ... 678
field-beginning ... 505
field-end ... 505
field-string ... 505
field-string-no-properties ... 505
file-accessible-directory-p ... 241
file-attributes ... 245
file-chase-links ... 243
file-directory-p ... 242
file-equal-p ... 243
file-executable-p ... 240
file-exists-p ... 240
file-expand-wildcards ... 261
file-in-directory-p ... 243
file-local-copy ... 265
file-locked-p ... 239
file-modes ... 244
file-modes-symbolic-to-number ... 250
file-name-absolute-p ... 253
file-name-all-completions ... 258
file-name-as-directory ... 254
file-name-base ... 253
file-name-completion ... 259
file-name-directory ... 252
file-name-extension ... 252
file-name-nondirectory ... 252
file-name-sans-extension ... 252
file-name-sans-versions ... 252
file-name-shadow-mode ... 69
file-newer-than-file-p ... 241
file-nlinks ... 244
file-ownership-preserved-p ... 241
file-readable-p ... 240
file-regular-p ... 242
file-relative-name ... 253
file-remote-p ... 265

file-selinux-context ..... 247
file-symlink-p ..... 242
file-truename ..... 243
file-writable-p ..... 241
fill-context-prefix ..... 477
fill-individual-paragraphs ..... 473
fill-paragraph ..... 473
fill-region ..... 473
fill-region-as-paragraph ..... 474
filter-buffer-substring ..... 456
find-buffer-visiting ..... 277
find-dired ..... 600
find-file ..... 231
find-file-at-point ..... 678
find-file-literally ..... 231
find-file-name-handler ..... 265
find-file-noselect ..... 231
find-file-other-window ..... 232
find-file-read-only ..... 232
find-font ..... 156
find-grep ..... 538
find-grep-dired ..... 600
find-image ..... 175
find-name-dired ..... 600
find-tag ..... 577
find-tag-other-frame ..... 578
find-tag-other-window ..... 578
find-tag-regexp ..... 578
fit-frame-to-buffer ..... 355
fit-window-to-buffer ..... 297
fixup-whitespace ..... 463
flush-lines ..... 215
flyspell-mode ..... 221
flyspell-prog-mode ..... 221
following-char ..... 454
font-at ..... 154
font-face-attributes ..... 156
font-family-list ..... 141
font-get ..... 156
font-lock-add-keywords ..... 434
font-lock-remove-keywords ..... 434
font-put ..... 155
font-spec ..... 155
font-xlfd-name ..... 156
fontp ..... 154
force-mode-line-update ..... 419
force-window-update ..... 112
format-find-file ..... 269
format-insert-file ..... 269
format-mode-line ..... 426
format-write-file ..... 269
fortune-to-signature ..... 631
forward-button ..... 183
forward-char ..... 59
forward-list ..... 522
forward-sexp ..... 521
forward-to-indentation ..... 487
forward-word ..... 59
frame-char-height ..... 354
frame-char-width ..... 354
frame-configuration-to-register ..... 107
frame-current-scroll-bars ..... 162
frame-first-window ..... 292
frame-height ..... 354
frame-list ..... 357
frame-live-p ..... 357
frame-parameter ..... 345
frame-parameters ..... 345
frame-pixel-height ..... 354
frame-pixel-width ..... 354
frame-pointer-visible-p ..... 363
frame-root-window ..... 290
frame-selected-window ..... 307
frame-terminal ..... 341
frame-visible-p ..... 360
frame-width ..... 354
framep ..... 341
fringe-bitmaps-at-pos ..... 160
fundamental-mode ..... 399

## G

gap-position ..... 288
gap-size ..... 288
gdb ..... 539
gdb-delete-breakpoint ..... 545
gdb-display-disassembly-for-thread ..... 546
gdb-display-locals-for-thread ..... 546
gdb-display-registers-for-thread ..... 546
gdb-display-stack-for-thread ..... 546
gdb-edit-value ..... 547
gdb-frames-select ..... 546
gdb-goto-breakpoint ..... 545
gdb-many-windows ..... 544
gdb-restore-windows ..... 544
gdb-select-thread ..... 545
gdb-toggle-breakpoint ..... 545
gdb-var-delete ..... 547
generate-new-buffer ..... 284
generate-new-buffer-name ..... 276
get-buffer ..... 275
get-buffer-create ..... 284
get-buffer-window ..... 310
get-buffer-window-list ..... 310
get-char-property ..... 489
get-char-property-and-overlay ..... 490
get-device-terminal ..... 343
get-file-buffer ..... 277
get-largest-window ..... 309
get-lru-window ..... 309
get-register ..... 508
get-text-property ..... 489
get-window-with-predicate ..... 309
getenv ..... 758
global-cwarn-mode ..... 533
global-set-key ..... 705

global-unset-key ..... 705
glyph-char ..... 192
glyph-face ..... 192
gnus ..... 651
gnus-group-exit ..... 652
gnus-group-kill-group ..... 652
gnus-group-list-all-groups ..... 652
gnus-group-list-groups ..... 652
gnus-group-next-unread-group ..... 652
gnus-group-prev-unread-group ..... 652
gnus-group-read-group ..... 652
gnus-group-unsubscribe-current-group ..... 652
gnus-summary-isearch-article ..... 652
gnus-summary-next-unread-article ..... 652
gnus-summary-prev-page ..... 652
gnus-summary-prev-unread-article ..... 652
gnus-summary-search-article-forward ..... 653
gomoku ..... 679
goto-address-at-point ..... 678
goto-address-mode ..... 678
goto-char ..... 60
goto-followup-to ..... 629
goto-line ..... 60
goto-reply-to ..... 629
grep ..... 538
grep-find ..... 538
gud-cont ..... 542
gud-def ..... 543
gud-down ..... 542
gud-finish ..... 542
gud-gdb ..... 539
gud-gdb-complete-command ..... 542
gud-jump ..... 542
gud-next ..... 541
gud-print ..... 541
gud-refresh ..... 541
gud-remove ..... 542
gud-step ..... 541
gud-stepi ..... 541
gud-tbreak ..... 542
gud-tooltip-mode ..... 540
gud-until ..... 542
gud-up ..... 542
gud-watch ..... 547

## H

handle-switch-frame ..... 359
handwrite ..... 671
hanoi ..... 679
help-buffer ..... 87
help-command ..... 85
help-setup-xref ..... 87
help-with-tutorial ..... 58
Helper-describe-bindings ..... 86
Helper-help ..... 87
hide-ifdef-mode ..... 533
holidays ..... 609
how-many ..... 215
hs-hide-all ..... 527
hs-hide-block ..... 527
hs-hide-level ..... 527
hs-minor-mode ..... 527
hs-show-all ..... 527
hs-show-block ..... 527
hs-show-region ..... 527
htmlfontify-buffer ..... 669

## I

icalendar-export-file,
icalendar-export-region ..... 621
icalendar-import-buffer ..... 621
icalendar-import-file ..... 621
icomplete-mode ..... 74
iconify-frame ..... 360
ielm ..... 552
image-animate ..... 178
image-animate-timer ..... 178
image-animated-p ..... 178
image-dired-display-thumbs ..... 601
image-flush ..... 178
image-load-path-for-library ..... 176
image-mask-p ..... 172
image-size ..... 177
image-type-available-p ..... 169
imagemagick-types ..... 173
imenu ..... 516
imenu-add-menubar-index ..... 516
imenu-add-to-menubar ..... 427
increment-register ..... 108
indent-according-to-mode ..... 484
indent-code-rigidly ..... 486, 518
indent-for-tab-command ..... 451, 484, 517
indent-line-function ..... 517
indent-pp-sexp ..... 518
indent-region ..... 452, 485
indent-relative ..... 451, 486
indent-relative-maybe ..... 487
indent-rigidly ..... 452, 485
indent-to ..... 483
indent-to-left-margin ..... 476
info-lookup-file ..... 526
info-lookup-symbol ..... 526
input-pending-p ..... 45
insert ..... 458
insert-abbrev-table-description ..... 582
insert-and-inherit ..... 501
insert-before-markers ..... 458
insert-before-markers-and-inherit ..... 501
insert-buffer ..... 459
insert-buffer-substring ..... 459
insert-buffer-substring-as-yank ..... 465
insert-buffer-substring-no-properties ... 459
insert-button ..... 181
insert-char ..... 58, 459

insert-directory ..... 261
insert-file-contents ..... 236
insert-file-contents-literally ..... 237
insert-for-yank ..... 465
insert-image ..... 176
insert-kbd-macro ..... 227
insert-register ..... 107, 508
insert-sliced-image ..... 177
insert-text-button ..... 181
interactive ..... 12
interactive-form ..... 14
invert-face ..... 145
invisible-p ..... 123
isearch-backward ..... 200
isearch-backward-regexp ..... 205
isearch-del-char ..... 202
isearch-forward ..... 199
isearch-forward-regexp ..... 205
isearch-forward-symbol ..... 204
isearch-forward-word ..... 204
isearch-toggle-input-method ..... 201
isearch-toggle-lax-whitespace ..... 201
isearch-toggle-specified-input-method ... 201
isearch-yank-char ..... 202
isearch-yank-kill ..... 202
isearch-yank-line ..... 202
isearch-yank-pop ..... 202
isearch-yank-word-or-char ..... 202
ispell ..... 220
ispell-buffer ..... 220
ispell-change-dictionary ..... 221
ispell-complete-word ..... 221
ispell-kill-ispell ..... 221
ispell-message ..... 630
ispell-region ..... 220
ispell-word ..... 220

## J

jdb ..... 539
jit-lock-register ..... 435
jit-lock-unregister ..... 436
jump-to-register ..... 106
just-one-space ..... 96, 463
justify-current-line ..... 474

## K

kbd ..... 706
kbd-macro-query ..... 226
keep-lines ..... 215
key-description ..... 84
keyboard-escape-quit ..... 718
keyboard-quit ..... 49, 717
keyboard-translate ..... 43
kill-append ..... 468
kill-buffer ..... 285
kill-compilation ..... 535
kill-emacs ..... 57
kill-line ..... 96
kill-local-variable ..... 698
kill-new ..... 468
kill-rectangle ..... 104
kill-region ..... 97, 465
kill-ring-save ..... 97
kill-sexp ..... 521
kill-whole-line ..... 96
kmacro-add-counter ..... 225
kmacro-bind-to-key ..... 227
kmacro-cycle-ring-next ..... 224
kmacro-cycle-ring-previous ..... 224
kmacro-edit-lossage ..... 228
kmacro-edit-macro ..... 228
kmacro-end-and-call-macro ..... 222
kmacro-end-macro ..... 223
kmacro-end-or-call-macro ..... 222
kmacro-end-or-call-macro-repeat ..... 224
kmacro-insert-counter ..... 225
kmacro-name-last-macro ..... 227
kmacro-set-counter ..... 225
kmacro-set-format ..... 225
kmacro-start-macro ..... 223
kmacro-start-macro-or-insert-counter ..... 222
kmacro-step-edit-macro ..... 228

## L

landmark ..... 680
last-buffer ..... 283
left-char ..... 59
left-word ..... 60
lgrep ..... 538
libxml-parse-html-region ..... 511
libxml-parse-xml-region ..... 511
life ..... 680
lisp-eval-defun ..... 552
lisp-interaction-mode ..... 552
list-bookmarks ..... 109
list-character-sets ..... 394
list-charset-chars ..... 394
list-coding-systems ..... 382
list-command-history ..... 76
list-fonts ..... 156
list-holidays ..... 609
list-input-methods ..... 381
list-matching-lines ..... 215
list-tags ..... 579
listify-key-sequence ..... 45
load ..... 549
load-file ..... 549
load-library ..... 549
load-theme ..... 693
local-set-key ..... 705
local-unset-key ..... 705
locate ..... 600
locate-file ..... 247

locate-user-emacs-file ..... 259
locate-with-filter ..... 600
lock-buffer ..... 239
log-edit-done ..... 560
log-edit-insert-changelog ..... 561
log-edit-show-diff ..... 560
log-edit-show-files ..... 560
log-view-toggle-entry-display ..... 564
lower-frame ..... 361
lpr-buffer ..... 669
lpr-region ..... 669
lunar-phases ..... 610
lwarn ..... 119

## M

mail-abbrev-insert-alias ..... 627
mail-add-attachment ..... 630
mail-fill-yanked-message ..... 629
mail-text ..... 629
make-abbrev-table ..... 581
make-button ..... 181
make-directory ..... 262
make-display-table ..... 190
make-frame ..... 342
make-frame-invisible ..... 361
make-frame-on-display ..... 344
make-frame-visible ..... 361
make-glyph-code ..... 192
make-help-screen ..... 87
make-indirect-buffer ..... 286
make-local-variable ..... 697
make-overlay ..... 128
make-progress-reporter ..... 116
make-symbolic-link ..... 249
make-temp-file ..... 257
make-temp-name ..... 258
make-text-button ..... 181
make-variable-buffer-local ..... 697
manual-entry ..... 526
mark-defun ..... 516
mark-sexp ..... 91, 521
mark-whole-buffer ..... 91
mark-word ..... 91
maximize-window ..... 297
md5 ..... 510
menu-bar-open ..... 9
merge-face-attribute ..... 144
message ..... 114
message-box ..... 115
message-goto-bcc ..... 629
message-goto-cc ..... 629
message-goto-fcc ..... 629
message-goto-subject ..... 629
message-goto-to ..... 629
message-insert-signature ..... 631
message-or-box ..... 114
message-send ..... 627
message-send-and-exit ..... 627
message-tab ..... 629
message-yank-original ..... 629
message-yank-prefix ..... 629
minibuffer-complete ..... 71
minibuffer-complete-and-exit ..... 72
minibuffer-complete-word ..... 71
minibuffer-electric-default-mode ..... 68
minibuffer-inactive-mode ..... 70
minimize-window ..... 297
mkdir ..... 262
mml-attach-file ..... 630
mode, Glasses ..... 528
modify-all-frames-parameters ..... 345
modify-frame-parameters ..... 345
momentary-string-display ..... 127
morse-region ..... 680
mouse-choose-completion ..... 71
mouse-movement-p ..... 34
mouse-on-link-p ..... 504
mouse-pixel-position ..... 363
mouse-position ..... 362
mouse-secondary-save-then-kill ..... 101
mouse-set-secondary ..... 101
mouse-start-secondary ..... 101
mouse-yank-secondary ..... 101
move-beginning-of-line ..... 59
move-end-of-line ..... 59
move-overlay ..... 129
move-to-column ..... 60, 483
move-to-left-margin ..... 476
move-to-window-line-top-bottom ..... 60
mpuz ..... 679
multi-occur ..... 215
multi-occur-in-matching-buffers ..... 215

## N

nato-region ..... 680
negative-argument ..... 51, 65
newline ..... 460
newline-and-indent ..... 451, 484, 517
next-button ..... 183
next-char-property-change ..... 493
next-completion ..... 72
next-error ..... 536
next-error-follow-minor-mode ..... 537
next-file ..... 579
next-frame ..... 357
next-history-element ..... 75
next-line ..... 59
next-matching-history-element ..... 75
next-overlay-change ..... 135
next-property-change ..... 492
next-single-char-property-change ..... 493
next-single-property-change ..... 493
next-window ..... 307
normal-erase-is-backspace-mode ..... 719

normal-mode ..... 403
not-modified ..... 279
number-to-register ..... 108

## O

occur ..... 215
one-window-p ..... 308
open-dribble-file ..... 726
open-line ..... 63
open-rectangle ..... 104
open-termscript ..... 726
other-buffer ..... 282
other-window ..... 308
overlay-buffer ..... 129
overlay-end ..... 129
overlay-get ..... 131
overlay-properties ..... 131
overlay-put ..... 131
overlay-recenter ..... 131
overlay-start ..... 129
overlayp ..... 128
overlays-at ..... 134
overlays-in ..... 135

## P

package-initialize ..... 682
package-upload-buffer ..... 685
package-upload-file ..... 685
partial completion ..... 73
pdb ..... 539
perldb ..... 539
point-to-register ..... 106
pong ..... 680
pop-global-mark ..... 93
pop-tag-mark ..... 578
pop-to-buffer ..... 312
pos-visible-in-window-p ..... 326
posn-actual-col-row ..... 36
posn-area ..... 35
posn-at-point ..... 37
posn-at-x-y ..... 37
posn-col-row ..... 36
posn-image ..... 36
posn-object ..... 36
posn-object-width-height ..... 36
posn-object-x-y ..... 36
posn-point ..... 35
posn-string ..... 36
posn-timestamp ..... 36
posn-window ..... 35
posn-x-y ..... 35
posnp ..... 35
pr-interface ..... 672
preceding-char ..... 455
prefer-coding-system ..... 383
prefix-numeric-value ..... 50
prepare-change-group ..... 511
prepend-to-buffer ..... 102
prepend-to-register ..... 107
previous-button ..... 183
previous-char-property-change ..... 493
previous-completion ..... 72
previous-frame ..... 358
previous-history-element ..... 75
previous-line ..... 59
previous-matching-history-element ..... 75
previous-overlay-change ..... 135
previous-property-change ..... 493
previous-single-char-property-change ..... 494
previous-single-property-change ..... 493
previous-window ..... 308
primitive-undo ..... 471
print-buffer ..... 669
print-buffer (MS-DOS) ..... 788
print-region ..... 669
print-region (MS-DOS) ..... 788
prog-mode ..... 407
progress-reporter-done ..... 117
progress-reporter-force-update ..... 116
progress-reporter-update ..... 116
propertize ..... 492
ps-despool ..... 670
ps-print-buffer ..... 670
ps-print-buffer (MS-DOS) ..... 788
ps-print-buffer-with-faces ..... 670
ps-print-region ..... 670
ps-print-region-with-faces ..... 670
ps-spool-buffer ..... 670
ps-spool-buffer (MS-DOS) ..... 788
ps-spool-buffer-with-faces ..... 670
ps-spool-region ..... 670
ps-spool-region-with-faces ..... 670
push-button ..... 183
put-image ..... 177
put-text-property ..... 490

## Q

quail-set-keyboard-layout ..... 381
quail-show-key ..... 381
query-replace ..... 213
query-replace-regexp ..... 213
quietly-read-abbrev-file ..... 583
quit-restore-window ..... 322
quit-window ..... 322, 588
quoted-insert ..... 58

## R

raise-frame ..... 361
re-search-backward ..... 205
re-search-forward ..... 205
read-char ..... 41
read-char-choice ..... 42

read-char-exclusive ..... 42
read-event ..... 41
read-file-modes ..... 250
read-kbd-macro ..... 85
read-key ..... 42
read-key-sequence ..... 39
read-key-sequence-vector ..... 40
read-only-mode ..... 281
read-quoted-char ..... 44
recenter ..... 329
recenter-top-bottom ..... 330
recode-file-name ..... 388
recode-region ..... 387
recompile ..... 535
rectangle ..... 104
recursion-depth ..... 53
recursive-edit ..... 52
redirect-frame-focus ..... 360
redisplay ..... 111
redraw-display ..... 111
redraw-frame ..... 111
reindent-then-newline-and-indent ..... 485
remove-from-invisibility-spec ..... 122
remove-hook ..... 398, 697
remove-images ..... 177
remove-list-of-text-properties ..... 491
remove-overlays ..... 129
remove-text-properties ..... 491
remove-untranslated-filesystem ..... 783
rename-buffer ..... 275
rename-file ..... 249
repeat ..... 66
repeat-complex-command ..... 76
replace-buffer-in-windows ..... 311
replace-regexp ..... 212
replace-string ..... 211
report-emacs-bug ..... 725
restore-buffer-modified-p ..... 278
revert-buffer (Dired) ..... 599
revert-buffer-with-coding-system ..... 387
rgrep ..... 538
right-char ..... 59
right-word ..... 59
rmail ..... 632
rmail-add-label ..... 639
rmail-beginning-of-message ..... 633
rmail-bury ..... 632
rmail-continue ..... 641
rmail-delete-backward ..... 634
rmail-delete-forward ..... 634
rmail-edit-current-message ..... 647
rmail-end-of-message ..... 633
rmail-epa-decrypt ..... 646
rmail-expunge ..... 635
rmail-expunge-and-save ..... 632
rmail-first-message ..... 634
rmail-forward ..... 641
rmail-get-new-mail ..... 637
rmail-input ..... 636
rmail-kill-label ..... 639
rmail-last-message ..... 634
rmail-mail ..... 641
rmail-mime ..... 646
rmail-mime-next-item ..... 646
rmail-mime-previous-item ..... 646
rmail-mime-toggle-hidden ..... 646
rmail-mode ..... 632
rmail-next-labeled-message ..... 639
rmail-next-message ..... 633
rmail-next-same-subject ..... 634
rmail-next-undeleted-message ..... 633
rmail-output ..... 637
rmail-output-as-seen ..... 637
rmail-output-body-to-file ..... 637
rmail-previous-labeled-message ..... 639
rmail-previous-message ..... 633
rmail-previous-same-subject ..... 634
rmail-previous-undeleted-message ..... 633
rmail-quit ..... 632
rmail-redecode-body ..... 647
rmail-reply ..... 640
rmail-resend ..... 641
rmail-retry-failure ..... 641
rmail-search ..... 634
rmail-show-message ..... 634
rmail-sort-by-author ..... 645
rmail-sort-by-correspondent ..... 645
rmail-sort-by-date ..... 645
rmail-sort-by-labels ..... 645
rmail-sort-by-lines ..... 645
rmail-sort-by-recipient ..... 645
rmail-sort-by-subject ..... 645
rmail-summary ..... 642
rmail-summary-bury ..... 644
rmail-summary-by-labels ..... 642
rmail-summary-by-recipients ..... 642
rmail-summary-by-regexp ..... 643
rmail-summary-by-senders ..... 643
rmail-summary-by-topic ..... 643
rmail-summary-quit ..... 644
rmail-summary-undelete-many ..... 643
rmail-summary-wipe ..... 644
rmail-toggle-header ..... 646
rmail-undelete-previous-message ..... 635
rot13-other-window ..... 648
run-hook-with-args ..... 397
run-hook-with-args-until-failure ..... 397
run-hook-with-args-until-success ..... 397
run-hook-wrapped ..... 398
run-hooks ..... 397
run-lisp ..... 552
run-mode-hooks ..... 408
run-scheme ..... 552

## S

same-window-p ... 319
save-buffer ... 234
save-buffers-kill-terminal ... 57
save-current-buffer ... 274
save-selected-window ... 306
save-some-buffers ... 234
save-window-excursion ... 336
scheme-mode ... 552
scroll-all-mode ... 676
scroll-bar-event-ratio ... 37
scroll-bar-scale ... 37
scroll-down ... 327
scroll-down-command ... 328
scroll-left ... 332
scroll-other-window ... 328
scroll-right ... 332
scroll-up ... 327
scroll-up-command ... 327
sdb ... 540
search-backward ... 203
search-forward ... 203
secure-hash ... 510
select-frame ... 359
select-frame-set-input-focus ... 359
select-window ... 306
selected-frame ... 358
selected-window ... 290
self-insert-command ... 460
send-invisible ... 658
serial-term ... 665
server-edit ... 666
server-eval-at ... 665
server-start ... 665
set-auto-mode ... 403
set-buffer ... 273
set-buffer-file-coding-system ... 386
set-buffer-major-mode ... 404
set-buffer-modified-p ... 278
set-buffer-process-coding-system ... 387
set-default-file-modes ... 250
set-display-table-slot ... 191
set-face-attribute ... 143
set-face-background ... 144
set-face-bold-p ... 145
set-face-font ... 145
set-face-foreground ... 144
set-face-inverse-video-p ... 145
set-face-italic-p ... 145
set-face-stipple ... 145
set-face-underline ... 145
set-file-modes ... 249
set-file-name-coding-system ... 388
set-file-selinux-context ... 251
set-file-times ... 251
set-fontset-font ... 153, 392
set-frame-configuration ... 362
set-frame-height ... 355
set-frame-parameter ... 345
set-frame-position ... 354
set-frame-selected-window ... 307
set-frame-size ... 354
set-frame-width ... 355
set-fringe-bitmap-face ... 161
set-goal-column ... 60
set-input-method ... 380
set-keyboard-coding-system ... 389
set-language-environment ... 377
set-left-margin ... 476
set-locale-environment ... 377
set-mark-command ... 89
set-mouse-pixel-position ... 363
set-mouse-position ... 363
set-next-selection-coding-system ... 387
set-register ... 508
set-right-margin ... 476
set-selection-coding-system ... 387
set-terminal-coding-system ... 389
set-terminal-parameter ... 356
set-text-properties ... 491
set-variable ... 695
set-visited-file-modtime ... 280
set-visited-file-name ... 277
set-window-buffer ... 309
set-window-combination-limit ... 304
set-window-configuration ... 335
set-window-dedicated-p ... 321
set-window-display-table ... 192
set-window-fringes ... 157
set-window-hscroll ... 332
set-window-margins ... 168
set-window-next-buffers ... 320
set-window-parameter ... 337
set-window-point ... 324
set-window-prev-buffers ... 319
set-window-scroll-bars ... 162
set-window-start ... 325
set-window-vscroll ... 330
setenv ... 758
setq-default ... 698
shell ... 656
shell-backward-command ... 658
shell-command ... 655
shell-command-on-region ... 656
shell-forward-command ... 658
shell-pushd-dextract ... 663
shell-pushd-dunique ... 663
shell-pushd-tohome ... 663
show-paren-mode ... 523
shrink-window-if-larger-than-buffer ... 297
single-key-description ... 84
sit-for ... 47
sleep-for ... 47
smie-bnf->prec2 ... 442
smie-close-block ... 442
smie-down-list ... 442

smie-merge-prec2s ..... 442
smie-prec2->grammar ..... 442
smie-precs->prec2 ..... 442
smie-rule-bolp ..... 447
smie-rule-hanging-p ..... 447
smie-rule-next-p ..... 447
smie-rule-parent ..... 448
smie-rule-parent-p ..... 447
smie-rule-prev-p ..... 447
smie-rule-separator ..... 448
smie-rule-sibling-p ..... 448
smie-setup ..... 441
snake ..... 680
Snarf-documentation ..... 82
solitaire ..... 680
sort-columns ..... 482, 674
sort-fields ..... 481, 672
sort-lines ..... 481, 672
sort-numeric-fields ..... 482, 672
sort-pages ..... 481, 672
sort-paragraphs ..... 481, 672
sort-regexp-fields ..... 480
sort-subr ..... 479
special-mode ..... 408
split-line ..... 451
split-window ..... 298
split-window-below ..... 299
split-window-right ..... 299
split-window-sensibly ..... 318
spook ..... 631
standard-display-8bit ..... 393
string-insert-rectangle ..... 104
string-rectangle ..... 104
string-width ..... 135
subst-char-in-region ..... 507
substitute-command-keys ..... 83
substitute-in-file-name ..... 256
substitute-key-definition ..... 714
subword-mode ..... 532
sunrise-sunset ..... 609
suspend-frame ..... 57
switch-to-buffer ..... 311
switch-to-buffer-other-frame ..... 312
switch-to-buffer-other-window ..... 312
switch-to-completions ..... 71
switch-to-next-buffer ..... 320
switch-to-prev-buffer ..... 320

## T

tab-to-tab-stop ..... 451, 487
tabify ..... 453
tabulated-list-init-header ..... 410
tabulated-list-mode ..... 409
tabulated-list-print ..... 410
tags-apropos ..... 579
tags-loop-continue ..... 578
tags-query-replace ..... 578
tags-search ..... 578
term ..... 663
term-char-mode ..... 664
term-line-mode ..... 664
term-pager-toggle ..... 664
terminal-list ..... 343
terminal-live-p ..... 341
terminal-name ..... 343
terminal-parameter ..... 356
terminal-parameters ..... 356
tetris ..... 680
text-char-description ..... 84
text-mode ..... 407
text-properties-at ..... 490
text-property-any ..... 494
text-property-not-all ..... 494
thing-at-point ..... 457
this-command-keys ..... 22
this-command-keys-vector ..... 22
timeclock-change ..... 622
timeclock-in ..... 622
timeclock-modeline-display ..... 622
timeclock-out ..... 622
timeclock-reread-log ..... 622
timeclock-when-to-leave ..... 622
timeclock-workday-remaining ..... 622
tmm-menubar ..... 9
toggle-debug-on-error ..... 727
toggle-enable-multibyte-characters ..... 377
toggle-gdb-all-registers ..... 547
toggle-input-method ..... 380
top-level ..... 53, 718
tpu-edt-on ..... 677
track-mouse ..... 362
transient-mark-mode ..... 94
translate-region ..... 507
transpose-chars ..... 218
transpose-lines ..... 218
transpose-regions ..... 509
transpose-sexps ..... 521
transpose-words ..... 218
truncate-string-to-width ..... 135
tty-color-alist ..... 369
tty-color-approximate ..... 369
tty-color-clear ..... 369
tty-color-define ..... 369
tty-color-translate ..... 369
tty-top-frame ..... 361

## U

unbury-buffer ..... 283
uncomment-region ..... 524
undigestify-rmail-message ..... 648
undo ..... 217
undo-boundary ..... 471
undo-only ..... 217
unforward-rmail-message ..... 641

unhandled-file-name-directory ..... 266
universal-argument ..... 51, 66
universal-coding-system-argument ..... 386
unlock-buffer ..... 239
unmorse-region ..... 680
untabify ..... 453
upcase-region ..... 488
upcase-word ..... 489

## V

vc-annotate ..... 562
vc-diff ..... 562
vc-dir ..... 565
vc-dir-mark ..... 567
vc-dir-mark-all-files ..... 567
vc-next-action ..... 558
vc-print-log ..... 564
vc-print-root-log ..... 564
vc-pull ..... 568
vc-register ..... 561
vc-revert ..... 565
vc-revision-other-window ..... 562
vc-root-diff ..... 562
verify-visited-file-modtime ..... 279
vi-mode ..... 677
view-hello-file ..... 375
view-register ..... 106, 508
vip-mode ..... 677
viper-mode ..... 677
visible-frame-list ..... 357
visit-tags-table ..... 576
visited-file-modtime ..... 280

## W

walk-windows ..... 308
warn ..... 119
wdired-change-to-wdired-mode ..... 601
wdired-finish-edit ..... 601
what-cursor-position ..... 65, 375
what-line ..... 64
which-function-mode ..... 517
while-no-input ..... 46
widget-backward ..... 687
widget-complete ..... 688
widget-forward ..... 687
window-absolute-pixel-edges ..... 335
window-at ..... 334
window-body-height ..... 294
window-body-size ..... 295
window-body-width ..... 294
window-buffer ..... 309
window-child ..... 292
window-combination-limit ..... 304
window-combined-p ..... 292
window-configuration-frame ..... 336
window-configuration-p ..... 336
window-configuration-to-register ..... 107
window-current-scroll-bars ..... 163
window-dedicated-p ..... 321
window-display-table ..... 192
window-edges ..... 333
window-end ..... 324
window-frame ..... 290
window-fringes ..... 157
window-full-height-p ..... 294
window-full-width-p ..... 294
window-hscroll ..... 332
window-in-direction ..... 292
window-inside-absolute-pixel-edges ..... 335
window-inside-edges ..... 333
window-inside-pixel-edges ..... 335
window-left-child ..... 292
window-left-column ..... 333
window-line-height ..... 326
window-list ..... 290
window-live-p ..... 289
window-margins ..... 168
window-next-buffers ..... 319
window-next-sibling ..... 292
window-parameter ..... 337
window-parameters ..... 337
window-parent ..... 291
window-pixel-edges ..... 334
window-point ..... 323
window-prev-buffers ..... 319
window-prev-sibling ..... 292
window-resizable ..... 296
window-resize ..... 296
window-scroll-bars ..... 162
window-size-fixed-p ..... 295
window-start ..... 324
window-state-get ..... 337
window-state-put ..... 337
window-system ..... 195
window-text-height ..... 295
window-top-child ..... 291
window-top-line ..... 333
window-total-height ..... 294
window-total-size ..... 294
window-total-width ..... 294
window-tree ..... 293
window-valid-p ..... 289
window-vscroll ..... 330
windowp ..... 289
with-current-buffer ..... 274
with-help-window ..... 87
with-local-quit ..... 49
with-output-to-temp-buffer ..... 125
with-selected-window ..... 307
with-temp-buffer ..... 274
with-temp-buffer-window ..... 127
with-temp-file ..... 238
with-temp-message ..... 114
with-wrapper-hook ..... 397

woman ..... 527
word-search-backward ..... 204
word-search-forward ..... 204
wordstar-mode ..... 677
write-abbrev-file ..... 583
write-file ..... 234
write-region ..... 237

## X

x-close-connection ..... 344
x-color-defined-p ..... 367
x-color-values ..... 368
x-defined-colors ..... 368
x-display-color-p ..... 371
x-display-list ..... 344
x-family-fonts ..... 152
x-get-resource ..... 370
x-get-selection ..... 366
x-list-fonts ..... 152
x-open-connection ..... 344
x-parse-geometry ..... 355
x-popup-dialog ..... 365
x-popup-menu ..... 363
x-server-vendor ..... 373
x-server-version ..... 373
x-set-selection ..... 366
xdb ..... 540

## Y

yank ..... 98, 466
yank-pop ..... 98, 467
yank-rectangle ..... 104

## Z

zap-to-char ..... 97
zone ..... 680
zrgrep ..... 538

# Variable Index

## A

abbrev-all-caps ... 584
abbrev-expand-functions ... 585
abbrev-file-name ... 583
abbrev-minor-mode-table-alist ... 586
abbrev-start-location ... 584
abbrev-start-location-buffer ... 585
abbrev-table-name-list ... 582
abbrevs-changed ... 583
adaptive-fill-first-line-regexp ... 478
adaptive-fill-function ... 478
adaptive-fill-mode ... 477
adaptive-fill-regexp ... 478
add-log-always-start-new-record ... 570
add-log-keep-changes-together ... 570
after-change-functions ... 513
after-change-major-mode-hook ... 409
after-insert-file-functions ... 271
after-make-frame-functions ... 342
after-save-hook ... 236
alpha, a frame parameter ... 353
appt-audible ... 620
appt-delete-window-function ... 620
appt-disp-window-function ... 620
appt-display-diary ... 620
appt-display-duration ... 620
appt-display-format ... 620
appt-display-mode-line ... 620
appt-message-warning-time ... 620
appt-warning-time-regexp ... 620
async-shell-command-buffer ... 656
auto-coding-alist ... 384
auto-coding-functions ... 384
auto-coding-regexp-alist ... 384
auto-fill-chars ... 478
auto-fill-function ... 478
auto-hscroll-mode ... 331
auto-lower, a frame parameter ... 350
auto-mode-alist ... 404
auto-raise, a frame parameter ... 350
auto-revert-check-vc-info ... 558
auto-window-vscroll ... 331

## B

background-color, a frame parameter ... 353
background-mode, a frame parameter ... 352
backward-delete-char-untabify-method ... 462
bdf-directory-list ... 672
before-change-functions ... 512
before-make-frame-hook ... 342
before-save-hook ... 236
bidi-display-reordering ... 195, 395
bidi-paragraph-direction ... 196, 395
blink-cursor-alist ... 352
blink-matching-delay ... 188, 522
blink-matching-paren ... 188, 522
blink-matching-paren-distance ... 188, 522
blink-paren-function ... 188
bookmark-save-flag ... 109
bookmark-search-size ... 109
border-color, a frame parameter ... 353
border-width, a frame parameter ... 349
browse-url-browser-function ... 678
browse-url-mailto-function ... 678
buffer-access-fontified-property ... 501
buffer-access-fontify-functions ... 501
buffer-auto-save-file-format ... 269
buffer-display-count ... 310
buffer-display-table ... 192
buffer-display-time ... 310
buffer-file-coding-system ... 385
buffer-file-format ... 269
buffer-file-name ... 276
buffer-file-number ... 277
buffer-file-truename ... 276
buffer-invisibility-spec ... 122
buffer-list, a frame parameter ... 350
buffer-offer-save ... 286
buffer-predicate, a frame parameter ... 350
buffer-read-only ... 281
buffer-save-without-query ... 286
buffer-substring-filters ... 457
buffer-undo-list ... 470

## C

c-default-style ... 520
c-hungry-delete-key ... 531
c-mode-hook ... 514
c-tab-always-indent ... 519
cache-long-line-scans ... 113
cal-html-css-default ... 607
calendar-date-style ... 617
calendar-daylight-savings-ends ... 621
calendar-daylight-savings-ends-time ... 622
calendar-daylight-savings-starts ... 621
calendar-daylight-time-offset ... 622
calendar-daylight-time-zone-name ... 610
calendar-latitude ... 610
calendar-location-name ... 610
calendar-longitude ... 610
calendar-mark-diary-entries-flag ... 616
calendar-mark-holidays-flag ... 609
calendar-remove-frame-by-deleting ... 607
calendar-standard-time-zone-name ... 610
calendar-time-zone ... 610
calendar-view-diary-initially-flag ... 616
calendar-view-holidays-initially-flag ... 609

calendar-week-start-day ..... 605
case-fold-search ..... 211
case-replace ..... 213
change-log-version-info-enabled ..... 570
change-log-version-number-regexp-list ... 570
change-major-mode-after-body-hook ..... 409
char-property-alias-alist ..... 490
coding ..... 385
comint-completion-addsuffix ..... 663
comint-completion-autolist ..... 663
comint-completion-fignore ..... 657
comint-completion-recexact ..... 663
comint-input-autoexpand ..... 662
comint-input-ignoredups ..... 663
comint-move-point-for-output ..... 662
comint-prompt-read-only ..... 662
comint-scroll-show-maximum-output ..... 662
comint-scroll-to-bottom-on-input ..... 662
comint-use-prompt-regexp ..... 659
command-history ..... 54, 76
command-line-args ..... 754
comment-column ..... 525
comment-end ..... 525
comment-indent-function ..... 525
comment-multi-line ..... 525
comment-padding ..... 525
comment-start ..... 525
comment-start-skip ..... 525
compilation-always-kill ..... 535
compilation-auto-jump-to-first-error ..... 535
compilation-context-lines ..... 536
compilation-environment ..... 535
compilation-error-regexp-alist ..... 536
compilation-scroll-output ..... 534
compilation-skip-threshold ..... 536
compile-command ..... 534
completion-auto-help ..... 74
completion-category-overrides ..... 73
completion-cycle-threshold ..... 74
completion-ignored-extensions ..... 74, 259
completion-styles ..... 73
confirm-kill-emacs ..... 57
confirm-nonexistent-file-or-buffer ..... 72
create-lockfiles ..... 239
crisp-override-meta-x ..... 676
ctl-arrow ..... 190
ctl-x-4-map ..... 704
ctl-x-map ..... 704
cua-enable-cua-keys ..... 105
cua-mode ..... 105
current-input-method ..... 380
current-language-environment ..... 377
current-prefix-arg ..... 51
cursor-color, a frame parameter ..... 353
cursor-in-echo-area ..... 118
cursor-in-non-selected-windows ..... 351
cursor-type ..... 351
cursor-type, a frame parameter ..... 351
custom-buffer-done-kill ..... 689
custom-enabled-themes ..... 693
custom-file ..... 690
custom-safe-themes ..... 693
custom-search-field ..... 687
custom-theme-directory ..... 692, 694
custom-theme-load-path ..... 692

## D

data-directory ..... 87
dbx-mode-hook ..... 543
debug-on-event ..... 727
debug-on-quit ..... 727
default-directory ..... 256
default-frame-alist ..... 346
default-input-method ..... 380
default-justification ..... 474
default-minibuffer-frame ..... 358
default-text-properties ..... 490
defining-kbd-macro ..... 55
delayed-warnings-hook ..... 121
delayed-warnings-list ..... 121
delete-active-region ..... 92
delete-by-moving-to-trash ..... 249, 262, 590
delete-frame-functions ..... 357
delete-terminal-functions ..... 343
desktop-buffer-mode-handlers ..... 450
desktop-clear-preserve-buffers-regexp ... 675
desktop-globals-to-clear ..... 675
desktop-path ..... 675
desktop-restore-eager ..... 675
desktop-save-buffer ..... 449
desktop-save-mode ..... 675
diary-file ..... 616
diary-mail-days ..... 616
diary-nonmarking-symbol ..... 617
diary-outlook-formats ..... 621
dired-auto-revert-buffer ..... 599
dired-chown-program ..... 594
dired-copy-preserve-time ..... 594
dired-dwim-target ..... 594
dired-garbage-files-regexp ..... 590
dired-isearch-filenames ..... 589
dired-kept-versions ..... 590
dired-listing-switches ..... 588
dired-recursive-copies ..... 594
dired-recursive-deletes ..... 590
dired-use-ls-dired ..... 588
dirtrack-list ..... 662
disable-point-adjustment ..... 23
disabled-command-function ..... 53
display, a frame parameter ..... 347
display-buffer-alist ..... 314
display-buffer-base-action ..... 314
display-buffer-fallback-action ..... 314
display-buffer-overriding-action ..... 314
display-mm-dimensions-alist ..... 372

display-type, a frame parameter ..... 347
dnd-protocol-alist ..... 367
doc-directory ..... 82
doc-view-cache-directory ..... 654
doc-view-continuous ..... 653
doc-view-resolution ..... 654
double-click-fuzz ..... 30, 710
double-click-time ..... 30, 710

## E

echo-area-clear-hook ..... 118
echo-keystrokes ..... 118
emacs-lisp-docstring-fill-column ..... 79
emacs-lisp-mode-hook ..... 514
enable-local-eval ..... 701
enable-local-variables ..... 701
enable-multibyte-characters ..... 376
enable-recursive-minibuffers ..... 70
esc-map ..... 704
eval-expression-debug-on-error ..... 551
eval-expression-print-length ..... 551
eval-expression-print-level ..... 551
exec-path ..... 655
executing-kbd-macro ..... 55
exit-language-environment-hook ..... 378
explicit-shell-file-name ..... 656
extra-keyboard-modifiers ..... 43

## F

face-font-family-alternatives ..... 150
face-font-registry-alternatives ..... 151
face-font-rescale-alist ..... 151
face-font-selection-order ..... 151
face-remapping-alist ..... 147
ff-related-file-alist ..... 533
file-coding-system-alist ..... 383
file-name-buffer-file-type-alist ..... 783
file-name-coding-system ..... 388
file-name-handler-alist ..... 263
file-precious-flag ..... 236
fill-column ..... 475
fill-forward-paragraph-function ..... 475
fill-individual-varying-indent ..... 474
fill-nobreak-predicate ..... 477
fill-paragraph-function ..... 475
fill-prefix ..... 475
filter-buffer-substring-functions ..... 456
find-file-hook ..... 232
find-file-literally ..... 233
find-file-not-found-functions ..... 232
find-file-wildcards ..... 232
find-ls-option ..... 600
find-tag-marker-ring-length ..... 578
first-change-hook ..... 513
fit-frame-to-buffer ..... 297
fit-frame-to-buffer-bottom-margin ..... 355
focus-follows-mouse ..... 360
font, a frame parameter ..... 353
font-backend, a frame parameter ..... 352
font-lock-beginning-of-syntax-function .. 438
font-lock-builtin-face ..... 437
font-lock-comment-delimiter-face ..... 437
font-lock-comment-face ..... 437
font-lock-constant-face ..... 437
font-lock-defaults ..... 429
font-lock-doc-face ..... 437
font-lock-extend-after-change-region-function ..... 440
font-lock-extra-managed-props ..... 435
font-lock-fontify-buffer-function ..... 435
font-lock-fontify-region-function ..... 435
font-lock-function-name-face ..... 437
font-lock-keyword-face ..... 437
font-lock-keywords ..... 430
font-lock-keywords-case-fold-search ..... 433
font-lock-keywords-only ..... 438
font-lock-mark-block-function ..... 435
font-lock-multiline ..... 439
font-lock-negation-char-face ..... 437
font-lock-preprocessor-face ..... 437
font-lock-string-face ..... 437
font-lock-syntactic-face-function ..... 438
font-lock-syntax-table ..... 438
font-lock-type-face ..... 437
font-lock-unfontify-buffer-function ..... 435
font-lock-unfontify-region-function ..... 435
font-lock-variable-name-face ..... 437
font-lock-warning-face ..... 437
font-slant-table (MS-Windows) ..... 789
font-weight-table (MS-Windows) ..... 789
fontification-functions ..... 149
foreground-color, a frame parameter ..... 353
format-alist ..... 267
frame-alpha-lower-limit ..... 353
frame-auto-hide-function ..... 323
frame-inherited-parameters ..... 342
frame-title-format ..... 356
fringe-cursor-alist ..... 159
fringe-indicator-alist ..... 158
fringes-outside-margins ..... 157
fullscreen, a frame parameter ..... 348
fundamental-mode-abbrev-table ..... 586

## G

gdb-delete-out-of-scope ..... 547
gdb-gud-control-all-threads ..... 548
gdb-many-windows ..... 544
gdb-mode-hook ..... 543
gdb-non-stop-setting ..... 548
gdb-show-changed-values ..... 547
gdb-show-threads-by-default ..... 545
gdb-speedbar-auto-raise ..... 548
gdb-stopped-functions ..... 548

gdb-switch-reasons ..... 548
gdb-switch-when-another-stopped ..... 548
gdb-thread-buffer-addresses ..... 546
gdb-thread-buffer-arguments ..... 545
gdb-thread-buffer-locations ..... 545
gdb-thread-buffer-verbose-names ..... 545
gdb-use-colon-colon-notation ..... 548
global-abbrev-table ..... 586
global-cwarn-mode ..... 533
global-disable-point-adjustment ..... 23
global-mark-ring-max ..... 93
global-mode-string ..... 423
glyphless-char-display ..... 192
glyphless-char-display-control ..... 193
grep-find-ignored-directories ..... 538
grep-regexp-alist ..... 536
gud-gdb-command-name ..... 543
gud-tooltip-echo-area ..... 540
gud-xdb-directories ..... 540

## H

header-line-format ..... 426
height, a frame parameter ..... 348
help-char ..... 86
help-enable-auto-load ..... 550
help-event-list ..... 86
help-form ..... 86
help-map ..... 85, 704
Helper-help-map ..... 87
hide-ifdef-shadow ..... 533
highlight-nonselected-windows ..... 89
history-delete-duplicates ..... 75
history-length ..... 75
hs-hide-comments-when-hiding-all ..... 528
hs-isearch-open ..... 528
hs-special-modes-alist ..... 528

## I

icon-left, a frame parameter ..... 347
icon-name, a frame parameter ..... 350
icon-title-format ..... 356
icon-top, a frame parameter ..... 348
icon-type, a frame parameter ..... 350
ignore-window-parameters ..... 338
image-cache-eviction-delay ..... 179
image-dired-external-viewer ..... 601
image-load-path ..... 176
image-types ..... 169
imagemagick-enabled-types ..... 174
imagemagick-types-inhibit ..... 174
imenu-auto-rescan ..... 516
imenu-case-fold-search ..... 428
imenu-create-index-function ..... 428
imenu-extract-index-name-function ..... 428
imenu-generic-expression ..... 427
imenu-prev-index-position-function ..... 428
imenu-sort-function ..... 516
imenu-syntax-alist ..... 428
indent-line-function ..... 484
indent-region-function ..... 485
indent-tabs-mode ..... 453, 484
indicate-buffer-boundaries ..... 158
indicate-empty-lines ..... 157
inferior-lisp-program ..... 552
inhibit-eol-conversion ..... 384
inhibit-file-name-handlers ..... 265
inhibit-file-name-operation ..... 265
inhibit-iso-escape-detection ..... 384
inhibit-local-variables-regexps ..... 403
inhibit-modification-hooks ..... 513
inhibit-point-motion-hooks ..... 499
inhibit-quit ..... 49
inhibit-read-only ..... 281
inhibit-startup-buffer-menu ..... 754
inhibit-startup-screen ..... 56, 757
inhibit-x-resources ..... 370
initial-environment ..... 758
initial-frame-alist ..... 345
initial-major-mode ..... 404
initial-scratch-message ..... 552
initial-window-system ..... 195
input-method-function ..... 44
input-method-highlight-flag ..... 380
input-method-verbose-flag ..... 380
insert-default-directory ..... 69
insert-directory-program ..... 262
internal-border-width, a frame parameter ..... 349
interpreter-mode-alist ..... 404
interprogram-cut-function ..... 468
interprogram-paste-function ..... 468
isearch-allow-scroll ..... 202
isearch-lazy-highlight ..... 200
isearch-mode-map ..... 202
isearch-resume-in-command-history ..... 76
ispell-complete-word-dict ..... 221
ispell-dictionary ..... 221
ispell-local-dictionary ..... 221
ispell-personal-dictionary ..... 221

## J

jdb-mode-hook ..... 543

## K

kbd-macro-termination-hook ..... 55
keyboard-coding-system ..... 389
keyboard-translate-table ..... 43
kill-buffer-hook ..... 285
kill-buffer-query-functions ..... 285
kill-do-not-save-duplicates ..... 97
kill-read-only-ok ..... 97, 465
kill-ring ..... 98, 469

kill-ring-max ..... 98, 469
kill-ring-yank-pointer ..... 469
kill-whole-line ..... 96
kmacro-ring-max ..... 224

## L

last-abbrev ..... 585
last-abbrev-location ..... 585
last-abbrev-text ..... 585
last-command ..... 20
last-command-event ..... 22
last-event-frame ..... 22
last-input-event ..... 45
last-kbd-macro ..... 55
last-nonmenu-event ..... 22
last-prefix-arg ..... 51
last-repeatable-command ..... 21
latin1-display ..... 392
left, a frame parameter ..... 347
left-fringe, a frame parameter ..... 349
left-fringe-width ..... 157
left-margin ..... 476
left-margin-width ..... 168
line-move-ignore-invisible ..... 123
line-move-visual ..... 61
line-prefix ..... 113
line-spacing ..... 137
line-spacing, a frame parameter ..... 349
lisp-body-indent ..... 519
lisp-indent-offset ..... 519
lisp-interaction-mode-hook ..... 514
lisp-mode-abbrev-table ..... 586
lisp-mode-hook ..... 514
list-buffers-directory ..... 278
load-dangerous-libraries ..... 550
load-path ..... 550
local-abbrev-table ..... 586
locale-charset-language-names ..... 377
locale-coding-system ..... 388
locale-language-names ..... 377
locale-preferred-coding-systems ..... 378
locate-command ..... 600
lpr-add-switches ..... 669
lpr-command (MS-DOS) ..... 788
lpr-commands ..... 669
lpr-headers-switches ..... 669
lpr-headers-switches (MS-DOS) ..... 788
lpr-printer-switch ..... 669
lpr-switches ..... 669
lpr-switches (MS-DOS) ..... 788

## M

magic-fallback-mode-alist ..... 404
magic-mode-alist ..... 404
mail-citation-hook ..... 629
mail-default-headers ..... 626
mail-dont-reply-to-names ..... 640
mail-from-style ..... 625
mail-mode-hook ..... 630
mail-personal-alias-file ..... 626
mail-setup-hook ..... 630
mail-signature ..... 630
mail-signature-file ..... 630
mail-user-agent ..... 631
major-mode ..... 399
Man-switches ..... 526
mark-even-if-inactive ..... 92
mark-ring-max ..... 93
max-image-size ..... 177
max-mini-window-height ..... 70
menu-bar-lines frame parameter ..... 349
message-kill-buffer-on-exit ..... 627
message-log-max ..... 7, 117
message-send-hook ..... 627
message-truncate-lines ..... 118
minibuffer, a frame parameter ..... 350
minibuffer-auto-raise ..... 361
minibuffer-eldef-shorten-default ..... 68
minibuffer-frame-alist ..... 346
minibuffer-local-completion-map ..... 704
minibuffer-local-filename-completion-map ..... 704
minibuffer-local-filename-must-match-map ..... 704
minibuffer-local-map ..... 704
minibuffer-local-must-match-map ..... 704
minibuffer-local-ns-map ..... 704
minor-mode-alist ..... 423
minor-mode-list ..... 413
mode-line-buffer-identification ..... 422
mode-line-client ..... 423
mode-line-format ..... 421
mode-line-frame-identification ..... 422
mode-line-modes ..... 422
mode-line-modified ..... 422
mode-line-mule-info ..... 422
mode-line-position ..... 422
mode-line-process ..... 423
mode-line-remote ..... 422
mode-name ..... 423
mode-specific-map ..... 704
mouse-1-click-follows-link ..... 503
mouse-color, a frame parameter ..... 353
mouse-position-function ..... 363
mouse-wheel-down-event ..... 31
mouse-wheel-up-event ..... 31
multiple-frames ..... 356

## N

name, a frame parameter ..... 347
next-error-highlight ..... 536
next-line-add-newlines ..... 61
next-screen-context-lines ..... 329

no-redraw-on-reenter ..... 111
normal-auto-fill-function ..... 478
normal-erase-is-backspace ..... 719
ns-pop-up-frames ..... 779
ns-standard-fontset-spec ..... 390
num-input-keys ..... 40
num-nonmacro-input-events ..... 42

## O

only-global-abbrevs ..... 583
open-paren-in-column-0-is-defun-start ... 515
other-window-scroll-buffer ..... 328
outer-window-id, a frame parameter ..... 351
overflow-newline-into-fringe ..... 159
overlay-arrow-position ..... 161
overlay-arrow-string ..... 161
overlay-arrow-variable-list ..... 162
overwrite-mode ..... 460

## P

package-archive-upload-base ..... 685
package-archives ..... 684
pdb-mode-hook ..... 543
perldb-mode-hook ..... 543
pop-up-frame-alist ..... 318
pop-up-frame-function ..... 318
pop-up-frames ..... 318
pop-up-windows ..... 317
post-command-hook ..... 11
post-self-insert-hook ..... 460
pre-command-hook ..... 11
prefix-arg ..... 51
prefix-help-command ..... 86
print-region-function (MS-DOS) ..... 788
printer-name ..... 669
printer-name, (MS-DOS/MS-Windows) ..... 787
prog-mode-hook ..... 407
ps-font-family ..... 671
ps-font-info-database ..... 671
ps-font-size ..... 671
ps-landscape-mode ..... 671
ps-lpr-command ..... 671
ps-lpr-command (MS-DOS) ..... 788
ps-lpr-switches ..... 671
ps-lpr-switches (MS-DOS) ..... 788
ps-multibyte-buffer ..... 671
ps-number-of-columns ..... 671
ps-page-dimensions-database ..... 671
ps-paper-type ..... 671
ps-print-color-p ..... 671
ps-print-header ..... 671
ps-printer-name ..... 671
ps-printer-name (MS-DOS) ..... 788
ps-use-face-background ..... 671

## Q

quit-flag ..... 49

## R

read-buffer-completion-ignore-case ..... 74
read-file-name-completion-ignore-case ..... 74
read-mail-command ..... 631
read-quoted-char-radix ..... 58
real-last-command ..... 21
recenter-positions ..... 330
recenter-redisplay ..... 330
redisplay-dont-pause ..... 111
redisplay-preemption-period ..... 112
register-alist ..... 507
register-separator ..... 107
remote-file-name-inhibit-cache ..... 266
replace-lax-whitespace ..... 211
require-final-newline ..... 236
resize-mini-windows ..... 70
right-fringe, a frame parameter ..... 349
right-fringe-width ..... 157
right-margin-width ..... 168
ring-bell-function ..... 194
rmail-automatic-folder-directives ..... 638
rmail-delete-after-output ..... 638
rmail-delete-message-hook ..... 635
rmail-displayed-headers ..... 646
rmail-edit-mode-hook ..... 647
rmail-enable-mime ..... 646
rmail-enable-mime-composing ..... 641
rmail-file-coding-system ..... 384
rmail-file-name ..... 632
rmail-highlighted-headers ..... 646
rmail-ignored-headers ..... 646
rmail-inbox-list ..... 637
rmail-mail-new-frame ..... 641
rmail-mode-hook ..... 632
rmail-movemail-flags ..... 650
rmail-movemail-program ..... 649
rmail-movemail-search-path ..... 649
rmail-nonignored-headers ..... 646
rmail-output-file-alist ..... 638
rmail-preserve-inbox ..... 636
rmail-primary-inbox-list ..... 635
rmail-redisplay-summary ..... 644
rmail-remote-password ..... 650
rmail-remote-password-required ..... 650
rmail-retry-ignored-headers ..... 641
rmail-secondary-file-directory ..... 637
rmail-secondary-file-regexp ..... 637
rmail-summary-line-count-flag ..... 643
rmail-summary-window-size ..... 643

## S

safe-local-eval-forms ..... 701
safe-local-variable-values ..... 701

same-window-buffer-names ..... 319
same-window-regexps ..... 319
save-abbrevs ..... 583
save-interprogram-paste-before-kill ..... 100
scalable-fonts-allowed ..... 151
scheme-mode-hook ..... 514
screen-gamma, a frame parameter ..... 352
scroll-bar-background, a frame parameter ..... 353
scroll-bar-foreground, a frame parameter ..... 353
scroll-bar-mode ..... 163
scroll-bar-width ..... 163
scroll-bar-width, a frame parameter ..... 349
scroll-conservatively ..... 328
scroll-down-aggressively ..... 328
scroll-error-top-bottom ..... 329
scroll-margin ..... 328
scroll-preserve-screen-position ..... 329
scroll-step ..... 329
scroll-up-aggressively ..... 329
sdb-mode-hook ..... 543
search-whitespace-regexp ..... 201
select-active-regions ..... 101
selection-coding-system ..... 366
selective-display ..... 124
selective-display-ellipses ..... 125
send-mail-function ..... 628
sendmail-coding-system ..... 385, 628
sentence-end-double-space ..... 475
sentence-end-without-period ..... 475
sentence-end-without-space ..... 475
server-auth-dir ..... 668
server-kill-new-buffers ..... 666
server-name ..... 665
server-port ..... 668
server-temp-file-regexp ..... 666
server-window ..... 666
set-language-environment-hook ..... 378
set-mark-command-repeat-pop ..... 93
shell-cd-regexp ..... 662
shell-command-default-error-buffer ..... 656
shell-command-regexp ..... 658
shell-completion-execonly ..... 663
shell-completion-fignore ..... 657
shell-file-name ..... 656
shell-input-ring-file-name ..... 661
shell-popd-regexp ..... 662
shell-prompt-pattern ..... 659
shell-pushd-regexp ..... 662
show-help-function ..... 499
small-temporary-file-directory ..... 258
sort-fold-case ..... 480, 674
sort-numeric-base ..... 482, 672
split-height-threshold ..... 318
split-width-threshold ..... 318
split-window-keep-point ..... 299
split-window-preferred-function ..... 317
standard-display-table ..... 192
standard-fontset-spec ..... 390
sticky, a frame parameter ..... 351
suggest-key-bindings ..... 78
switch-to-buffer-preserve-window-point ..... 312
switch-to-visible-buffer ..... 320

## T

tab-always-indent ..... 453, 485
tab-stop-list ..... 452, 487
tab-width ..... 190, 453
tabulated-list-entries ..... 409
tabulated-list-format ..... 409
tabulated-list-printer ..... 410
tabulated-list-revert-hook ..... 410
tabulated-list-sort-key ..... 410
tags-apropos-additional-actions ..... 579
tags-apropos-verbose ..... 579
tags-case-fold-search ..... 579
tags-file-name ..... 576
tags-table-list ..... 577
tags-tag-face ..... 579
temp-buffer-setup-hook ..... 126
temp-buffer-show-function ..... 126
temp-buffer-show-hook ..... 127
temp-buffer-window-setup-hook ..... 127
temp-buffer-window-show-hook ..... 127
temporary-file-directory ..... 257
term-file-prefix ..... 715
term-setup-hook ..... 715
text-mode-abbrev-table ..... 586
text-property-default-nonsticky ..... 500
this-command ..... 21
this-command-keys-shift-translated ..... 40
this-original-command ..... 21
three-step-help ..... 88
timeclock-ask-before-exiting ..... 622
timeclock-file ..... 622
timeclock-modeline-display ..... 622
title, a frame parameter ..... 347
tool-bar-lines frame parameter ..... 349
tool-bar-position frame parameter ..... 349
top, a frame parameter ..... 347
track-eol ..... 61
truncate-lines ..... 112
truncate-partial-width-windows ..... 112
tty-color-mode, a frame parameter ..... 352

## U

underline-minimum-offset ..... 141
undo-ask-before-discard ..... 472
undo-in-progress ..... 471
undo-limit ..... 218, 472
undo-outer-limit ..... 218, 472
undo-strong-limit ..... 218, 472

unibyte-display-via-language-environment ..... 393
unread-command-events ..... 45
unsplittable, a frame parameter ..... 350
use-hard-newlines ..... 475
user-full-name ..... 625
user-mail-address ..... 625, 713
user-mail-address, initialization ..... 759
user-position, a frame parameter ..... 348
user-size, a frame parameter ..... 348

## V

vc-diff-switches ..... 562
vc-directory-exclusion-list ..... 566
vc-log-mode-hook ..... 560
vc-log-show-limit ..... 565
vc-mode ..... 422
vc-revert-show-diff ..... 565
vertical-scroll-bar ..... 162
vertical-scroll-bars, a frame parameter .. 349
visibility, a frame parameter ..... 350
visible-bell ..... 194
void-text-area-pointer ..... 365

## W

w32-apps-modifier ..... 785
w32-charset-info-alist ..... 789
w32-get-true-file-attributes ..... 783
w32-lwindow-modifier ..... 785
w32-mouse-button-tolerance ..... 785
w32-pass-alt-to-system ..... 785
w32-pass-extra-mouse-buttons-to-system .. 786
w32-quote-process-args ..... 786
w32-rwindow-modifier ..... 785
w32-scroll-lock-modifier ..... 785
w32-standard-fontset-spec ..... 390
w32-swap-mouse-buttons ..... 786
w32-use-visible-system-caret ..... 790
wait-for-wm, a frame parameter ..... 351
warning-fill-prefix ..... 120
warning-levels ..... 119
warning-minimum-level ..... 120
warning-minimum-log-level ..... 120
warning-prefix-function ..... 120
warning-series ..... 120
warning-suppress-log-types ..... 121
warning-suppress-types ..... 121
warning-type-format ..... 120
which-func-modes ..... 517
width, a frame parameter ..... 348
window-combination-limit ..... 303
window-combination-resize ..... 304
window-configuration-change-hook ..... 340
window-id, a frame parameter ..... 350
window-min-height ..... 295
window-min-width ..... 295
window-persistent-parameters ..... 338
window-point-insertion-type ..... 324
window-scroll-functions ..... 339
window-setup-hook ..... 195
window-size-change-functions ..... 340
window-size-fixed ..... 295
window-system ..... 194
wrap-prefix ..... 113
write-contents-functions ..... 235
write-file-functions ..... 235
write-region-annotate-functions ..... 270
write-region-post-annotation-function ... 270

## X

x-bitmap-file-path ..... 141
x-dnd-known-types ..... 367
x-dnd-test-function ..... 367
x-dnd-types-alist ..... 367
x-pointer-shape ..... 365
x-resource-class ..... 370
x-resource-name ..... 370
x-select-enable-clipboard ..... 100
x-select-enable-clipboard-manager ..... 100
x-select-enable-primary ..... 100
x-select-request-type ..... 387
x-sensitive-text-pointer-shape ..... 366
xdb-mode-hook ..... 543

## Y

yank-excluded-properties ..... 466
yank-handled-properties ..... 466
yank-pop-change-selection ..... 100
yank-undo-function ..... 467

# Concept Index


## $

'$' in display ..... 112

## (

( in leftmost column ..... 515

## *

'*' in interactive ..... 13
'*Messages*' buffer ..... 7
'*scratch*' ..... 404

## -

–/—/.-./.../ ..... 680

## .

'.dir-locals.el' file ..... 701
.emacs file ..... 711
'.mailrc' file ..... 626
'.newsrc' file ..... 651
'.timelog' file ..... 622

## /

// in file name ..... 69

## ?

'?' in display ..... 375

## @

'@' in interactive ..... 13

## ^

'^' in interactive ..... 13

## _

'_emacs' init file, MS-Windows ..... 785

## \

'\' in display ..... 112

## ~

~/.emacs file ..... 711
'~/.emacs.d/gtkrc' file ..... 771
'~/.gtkrc-2.0' file ..... 771
'~/.Xdefaults' file ..... 769
'~/.Xresources' file ..... 769

## 8

8-bit display ..... 393
8-bit input ..... 393

## A

abbrev ..... 581
abbrev tables in modes ..... 401
abbreviated file names ..... 255
abnormal hook ..... 396, 696
aborting ..... 52
aborting recursive edit ..... 718
absolute file name ..... 253
accented characters ..... 393
accessibility of a file ..... 240
accumulating scattered text ..... 102
action alist, for display-buffer ..... 313
action function, for display-buffer ..... 313
action options (command line) ..... 754
active display table ..... 191
active region ..... 89
Adding to the kill ring in Dired ..... 602
addpm, MS-Windows installation program ..... 762
adjusting point ..... 23
advertised binding ..... 83
alarm clock ..... 620
alignment for comments ..... 523
Alt key invokes menu (Windows) ..... 785
ALTERNATE_EDITOR environment variable ..... 667
animate ..... 679
animation ..... 178
anonymous face ..... 137
appending kills in the ring ..... 99
appointment notification ..... 620
Arch ..... 555
arguments (command line) ..... 754
arguments to commands ..... 65
arguments, interactive entry ..... 12
arrow keys ..... 59
ASCII control characters ..... 189
Asm mode ..... 533
assembler mode ..... 533
astronomical day numbers ..... 611
atomic changes ..... 511
attribute (Rmail) ..... 638
attributes of text ..... 489
Auto Fill mode ..... 478
autoload ..... 550

autoload Lisp libraries ..... 714
automatic face assignment ..... 149
AWK mode ..... 530

## B

back end (version control) ..... 554
back reference, in regexp ..... 209
back reference, in regexp replacement ..... 212
background color, command-line argument .... 763
background mode, on `xterm` ..... 761
BACKSPACE vs DEL ..... 718
backtrace for bug reports ..... 728
Bahá'í calendar ..... 612
balanced expression ..... 521
balancing parentheses ..... 188
balancing window sizes ..... 297
base 64 encoding ..... 509
base buffer ..... 286
base direction of a paragraph ..... 196
base direction of paragraphs ..... 395
base location, package archive ..... 684
basic code (of input character) ..... 23
batch mode ..... 756
Bazaar ..... 555
before point, insertion ..... 458
bell ..... 194
bidirectional display ..... 195
bidirectional editing ..... 394
bidirectional reordering ..... 195
binary files, on MS-DOS/MS-Windows ..... 783
binding keyboard macros ..... 227
binding keys ..... 705
bitmaps, fringe ..... 159
blank lines ..... 63
blank lines in programs ..... 525
blinking cursor disable, command-line argument ..... 768
blinking parentheses ..... 188
body height of a window ..... 294
body of a window ..... 293
body size of a window ..... 294
body width of a window ..... 294
bookmarks ..... 109
border color, command-line argument ..... 764
borders (X Window System) ..... 766
boredom ..... 679
braces, moving across ..... 522
branch (version control) ..... 567
Brief emulation ..... 676
Browse-URL ..... 677
buffer ..... 272
buffer contents ..... 454
buffer definitions index ..... 516
buffer file name ..... 276
buffer list ..... 281
buffer modification ..... 278
buffer names ..... 275
buffer, read-only ..... 280
buffer-local hooks ..... 697
buffer-local variables in modes ..... 402
buffers without undo information ..... 275
buffers, controlled in windows ..... 309
buffers, creating ..... 284
buffers, killing ..... 284
bug tracker ..... 722
bugs ..... 722
building programs ..... 534
button buffer commands ..... 182
button down events ..... 709
button properties ..... 179
button types ..... 180
button-down event ..... 28
buttons (customization buffer) ..... 686
buttons in buffers ..... 179
bypassing init and '`default.el`' file ..... 757
byte code ..... 549
byte-compiling several files (in Dired) ..... 595
bzr ..... 555

## C

C editing ..... 514
C mode ..... 530
C++ class browser, tags ..... 572
C++ mode ..... 530
C-c C-c (Log Edit mode) ..... 560
`C-g` ..... 48
calendar ..... 604
calendar and HTML ..... 607
calendar and LaTeX ..... 607
calendar, first day of week ..... 605
call Lisp functions, command-line argument ... 755
camel case ..... 528
case conversion in buffers ..... 487
case in completion ..... 74
case-sensitivity and completion ..... 74
case-sensitivity and tags search ..... 579
categories of characters ..... 210
Celtic ..... 374
centering point ..... 329
centralized version control ..... 557
change Emacs directory ..... 756
change hooks ..... 512
change hooks for a character ..... 498
change log ..... 570
Change Log mode ..... 570
changes, undoing ..... 217
changeset-based version control ..... 556
changing file group (in Dired) ..... 594
changing file owner (in Dired) ..... 594
changing file permissions (in Dired) ..... 594
changing file time (in Dired) ..... 595
changing to another buffer ..... 272
changing window size ..... 295
char mode (terminal emulator) ..... 663

character insertion ... 460
character printing ... 84
character set of character at point ... 375
character syntax ... 713
characters for interactive codes ... 14
characters in a certain charset ... 394
characters which belong to a specific language ... 210
characters, inserting by name or code-point ... 58
charsets ... 394
checking out files ... 555
checking spelling ... 219
checking syntax ... 539
child window ... 290
Chinese ... 374
Chinese calendar ... 612
ciphers ... 679
citing mail ... 629
class browser, C++ ... 572
click event ... 26
click events ... 709
clickable buttons in buffers ... 179
clickable text ... 502
client frame ... 667
clipboard ... 100, 366
clipboard manager ... 100
clipboard support (for MS-Windows) ... 367
codes, interactive, description of ... 14
coding systems ... 381
color emulation on black-and-white printers ... 671
color names ... 367
color of window, from command line ... 763
color scheme ... 692
colors on text terminals ... 369
columns ... 482
columns (and rectangles) ... 103
columns (indentation) ... 451
Comint mode ... 659
`comint-highlight-input` face ... 656
`comint-highlight-prompt` face ... 656
command history ... 54, 76
command line arguments ... 754
command loop ... 11
command loop, recursive ... 51
commands, defining ... 12
comments ... 523
comments on customized settings ... 689
Common Lisp ... 552
compare files (in Dired) ... 597
comparing buffer text ... 457
comparing file modification time ... 279
compilation buffer, keeping point at end ... 534
compilation errors ... 534
Compilation mode ... 535
completion ... 70
completion (Lisp symbols) ... 528
completion (symbol names) ... 528, 579
completion alternative ... 70
completion list ... 71
completion style ... 72
completion, file name ... 258
complex command ... 54
compose character ... 393
`composition` property, and point display ... 23
compressing files (in Dired) ... 595
concatenating bidirectional strings ... 197
conditional selection of windows ... 309
Conf mode ... 514
confirming in the minibuffer ... 72
conflicts ... 569
connecting to remote host ... 664
content directory, package ... 681
continuation line ... 63
continuation lines ... 112
contributing to Emacs ... 731
control character printing ... 84
control characters in display ... 190
control characters, reading ... 44
controller part, model/view/controller ... 187
conventions for writing major modes ... 399
conventions for writing minor modes ... 413
converting file names from/to MS-Windows syntax ... 251
coordinate, relative to frame ... 333
Coptic calendar ... 611
copy ... 100
copying files ... 248
copying files (in Dired) ... 594
copying text ... 97
CORBA IDL mode ... 530
core dump ... 721
correcting spelling ... 219
counting columns ... 482
CPerl mode ... 514
crash report ... 720
creating buffers ... 284
creating, copying and deleting directories ... 262
CRiSP mode ... 676
cryptanalysis ... 679
cryptographic hash ... 510
CSSC ... 555
CUA key bindings ... 105
current buffer ... 272
current command ... 21
current function name in mode line ... 517
cursor ... 6, 323
cursor color, command-line argument ... 764
cursor location ... 64
cursor location, on MS-DOS ... 782
cursor motion ... 59
cursor position for `display` properties and overlays ... 497
cursor, and frame parameters ... 351
cursor, fringe ... 159
custom themes ... 692
custom themes, creating ... 693

customizable variable ... 686
customization ... 686
customization buffer ... 686
customization groups ... 686
customizing faces ... 690
customizing Lisp indentation ... 518
customizing variables ... 688
cut ... 100
cut and paste ... 804
cutting text ... 95
CVS ... 555
CWarn mode ... 533
cyclic ordering of windows ... 307
Cyrillic ... 374
Czech ... 374

## D

daemon, Emacs ... 665
day of year ... 606
daylight saving time ... 621
DBX ... 539
dead character ... 393
debbugs package ... 723
debuggers ... 539
debugging Emacs, tricks and techniques ... 729
decentralized version control ... 557
decoding file formats ... 267
decoding mail messages (Rmail) ... 647
decoding non-ASCII keyboard input on X ... 388
decrypting files (in Dired) ... 595
dedicated window ... 321
default argument ... 68
default argument string ... 14
default directory ... 68
'`default.el`' file, not loading ... 757
'`default.el`', the default init file ... 711
defining commands ... 12
defining keyboard macros ... 222
defuns ... 515
DEL vs BACKSPACE ... 718
Delete Selection mode ... 92
`delete-frame` event ... 31
deleting auto-save files ... 590
deleting blank lines ... 63
deleting characters and lines ... 61
deleting files ... 248
deleting files (in Dired) ... 589
deleting frames ... 357
deleting previous char ... 461
deleting some backup files ... 590
deleting text vs killing ... 460
deleting whitespace ... 462
deleting windows ... 300
deletion ... 95
deletion (Rmail) ... 634
Delphi mode ... 514
dependencies ... 681
derived mode ... 405
describe characters and events ... 84
description for interactive codes ... 14
desktop ... 675
desktop save mode ... 449
desktop shortcut, MS-Windows ... 781
Devanagari ... 374
device for Emacs terminal I/O ... 756
dialog boxes ... 364
diary ... 615
diary file ... 616
digest message ... 648
directory header lines ... 598
directory name ... 254
directory part (of file name) ... 251
directory tracking ... 662
directory where Emacs starts on MS-Windows ... 781
directory-local variables ... 701
directory-oriented functions ... 260
Dired ... 588
Dired and version control ... 602
Dired sorting ... 600
Dired, and MS-Windows/MS-DOS ... 784
Dirtrack mode ... 662
disable window system ... 756
disabled command ... 53, 711
disabling undo ... 472
discarding input ... 46
display action ... 313
`DISPLAY` environment variable ... 762
display feature testing ... 370
display for Emacs frame ... 756
display margins ... 167
display message in echo area ... 114
display name (X Window System) ... 762
display properties, and bidi reordering of text ... 196
`display` property, and point display ... 23
display specification ... 163
display table ... 190
display, abstract ... 183
display, arbitrary objects ... 183
displaying a buffer ... 311
displays, multiple ... 342
distributed version control ... 557
DNS mode ... 514
'`DOC-version`' (documentation) file ... 80
doc-view-minor-mode ... 653
document viewer (DocView) ... 653
documentation conventions ... 79
documentation for major mode ... 405
documentation strings ... 79
documentation, keys in ... 82
DocView mode ... 653
DOS applications, running from Emacs ... 786
DOS-style end-of-line display ... 384
DOS-to-Unix conversion of files ... 782

double clicks ... 709
double slash in file name ... 69
double-click events ... 29
down events ... 709
downcase file names ... 597
downcasing in `lookup-key` ... 40
drag and drop, Dired ... 603
drag event ... 28
drag events ... 709
`drag-n-drop` event ... 31
dribble file ... 726
DSSSL mode ... 514
Dutch ... 374
DVI file ... 653

## E

Ebrowse ... 572
echo area ... 7, 114
echo area message ... 7
echoing ... 7
EDE (Emacs Development Environment) ... 580
Edebug ... 727
editable fields (customization buffer) ... 686
editing binary files ... 674
editing level, recursive ... 675
editor command loop ... 11
`EDITOR` environment variable ... 665
EDT ... 676
Eldoc mode ... 527
Electric Indent mode ... 453
Electric Pair mode ... 523
Eliza ... 680
Emacs as a server ... 665
Emacs Development Environment ... 580
`EMACS` environment variable ... 657
Emacs event standard notation ... 84
Emacs icon, a gnu ... 767
Emacs initialization file ... 711
Emacs Lisp mode ... 550
`emacs-internal`, coding system ... 383
`EMACS_SERVER_FILE` environment variable ... 668
emacsclient ... 665
`emacsclient` invocation ... 666
`emacsclient` options ... 667
emacsclient, on MS-Windows ... 782
emacsclient.exe ... 781
emacsclientw.exe ... 781
email ... 624
emulating other editors ... 676
emulation of Brief ... 676
encoding file formats ... 267
encoding of characters ... 374
encrypted mails (reading in Rmail) ... 646
encrypting files (in Dired) ... 595
end-of-line convention, mode-line indication ... 8
end-of-line conversion ... 382
end-of-line conversion on MS-DOS/MS-Windows ... 782
entering Emacs ... 56
environment variables ... 758
environment variables for subshells ... 656
erasing characters and lines ... 61
error display ... 114
error log ... 534
error message ... 7
errors in init file ... 758
escape sequences in files ... 384
`ESHELL` environment variable ... 656
etags ... 571
`etags` program ... 574
Ethiopic ... 374
Ethiopic calendar ... 611
Euro sign ... 377
European character sets ... 393
evaluate expression, command-line argument ... 755
evaluated expression argument ... 16
event printing ... 84
event type ... 33
event, reading only one ... 41
events ... 23
ewoc ... 183
examining the `interactive` form ... 14
examining windows ... 309
examples of using `interactive` ... 17
execute with prefix argument ... 19
exit recursive editing ... 52
exiting ... 57
exiting recursive edit ... 675
expanding subdirectories in Dired ... 598
expansion of C macros ... 532
expansion of file names ... 255
expression ... 521
expunging (Rmail) ... 634

## F

face alias ... 148
face at point ... 375
face attributes ... 138
face codes of text ... 494
face name ... 137
face specification ... 141
faces ... 137
faces for font lock ... 436
faces for highlighting query replace ... 213
faces for highlighting search matches ... 199
faces, automatic choice ... 149
faces, customizing ... 690
Feedmail ... 628
FFAP minor mode ... 678
fields ... 504
file accessibility ... 240
file age ... 241
file attributes ... 244

file comparison (in Dired) ... 597
file database (locate) ... 600
file format conversion ... 267
file handler ... 263
file hard link ... 248
file local variables ... 698
file locks ... 239
file management ... 588
file mode specification error ... 403
file modes ... 244
file modes and MS-DOS ... 244
file modes, setting ... 249
file modification time ... 241
file name abbreviations ... 255
file name completion subroutines ... 258
file name of buffer ... 276
file name of directory ... 254
file names ... 251
file names in directory ... 260
file names on MS-Windows ... 783
file names with non-ASCII characters ... 388
file open error ... 233
file permissions ... 244
file permissions, setting ... 249
file symbolic links ... 242
file version in change log entries ... 570
file with multiple names ... 248
file, information about ... 240
file-based version control ... 556
file-name completion, on MS-Windows ... 783
filesets, VC ... 558
filling text ... 473
filling, automatic ... 478
`find` and Dired ... 600
find file in path ... 247
finding file at point ... 678
finding files ... 230
finding files containing regexp matches (in Dired) ... 593
finding strings within text ... 199
finding windows ... 308
fixed-size window ... 295
fixing incorrectly decoded mail messages ... 647
flagging files (in Dired) ... 589
flagging many files for deletion (in Dired) ... 590
flushing input ... 46
Flyspell mode ... 221
focus event ... 30
follow links ... 502
font and color, frame parameters ... 352
font antialiasing (MS Windows) ... 790
font backend selection (MS-Windows) ... 789
font lock faces ... 436
Font Lock mode ... 429
font name (X Window System) ... 763
font of character at point ... 375
font properties (MS Windows gdi backend) ... 790
font properties (MS Windows) ... 789
font scripts (MS Windows) ... 790
font specification (MS Windows) ... 788
font Unicode subranges (MS Windows) ... 790
fonts and faces ... 690
fonts for PostScript printing ... 671
fonts for various scripts ... 389
fontsets ... 389
fontsets, modifying ... 392
forcing redisplay ... 111
foreground color, command-line argument ... 763
format definition ... 267
fortune cookies ... 631
forwarding a message ... 641
frame ... 6, 341
frame configuration ... 362
frame layout parameters ... 349
frame parameters ... 345
frame parameters for windowed displays ... 346
frame size ... 354
frame title ... 356
frame title, command-line argument ... 767
frame visibility ... 360
frame-relative coordinate ... 333
frames, scanning all ... 357
French Revolutionary calendar ... 611
fringe bitmaps ... 159
fringe cursors ... 159
fringe indicators ... 157
fringes ... 156
fringes, and continuation lines ... 63
fringes, and empty line indication ... 157
fringes, for debugging ... 544
full-height window ... 294
full-screen frames ... 348
full-width window ... 294
fullheight, command-line argument ... 765
fullscreen, command-line argument ... 765
fullwidth, command-line argument ... 765
function key ... 703
function keys ... 24
function, move to beginning or end ... 516
functions in modes ... 400
functions, making them interactive ... 12

## G

gamma correction ... 352
GDB ... 539
GDB User Interface layout ... 544
generic mode ... 411
geometry of Emacs window ... 765
geometry specification ... 355
geometry, command-line argument ... 765
German ... 374
getting help with keys ... 63
Ghostscript, use for PostScript printing ... 788
GIF ... 173
git ... 555

Glasses mode ... 528
global keymap ... 703
global mark ... 105
global mark ring ... 93
global substitution ... 211
glyph ... 192
glyphless characters ... 192
GNU Arch ... 555
Gnus ... 651
GNUstep ... 778
Go Moku ... 679
Goto Address mode ... 678
graphic characters ... 58
graphical display ... 341
graphical terminal ... 341
Greek ... 374
Gregorian calendar ... 611
growing minibuffer ... 69
GTK styles ... 774
GTK widget classes ... 773
GTK widget names ... 772, 773
GTK+ resources ... 771
GUD interaction buffer ... 540
GUD library ... 539
GUD Tooltip mode ... 540

## H

handwriting ... 671
hard links (in Dired) ... 594
hardcopy ... 669
hash, cryptographic ... 510
header line (Dired) ... 598
header line (of a window) ... 426
`header-line` prefix key ... 40
headers (of mail message) ... 625
Hebrew ... 374
Hebrew calendar ... 611
height of a window ... 294
height of minibuffer ... 69
help for major mode ... 405
`help-echo` event ... 31
hex editing ... 674
Hexl mode ... 674
hg ... 555
hidden buffers ... 275
Hide-ifdef mode ... 533
Hideshow mode ... 527
hiding subdirectories (Dired) ... 599
highlighting matching parentheses ... 523
highlighting region ... 94
Hindi ... 374
history of commands ... 54, 76
history of minibuffer input ... 74
history reference ... 661
holidays ... 608
`HOME` directory on MS-Windows ... 784
home directory shorthand ... 69
hook ... 696
hooks ... 396
hooks for changing a character ... 498
hooks for motion of point ... 499
hooks for text changes ... 512
hooks for window operations ... 339
horizontal combination ... 291
horizontal position ... 482
horizontal scrolling ... 331
`horizontal-scroll-bar` prefix key ... 40
hungry deletion (C Mode) ... 531

## I

iCalendar support ... 621
Icomplete mode ... 74
Icon mode ... 514
iconified frame ... 360
`iconify-frame` event ... 31
iconifying ... 57
icons (X Window System) ... 767
IDL mode ... 530
ignored file names, in completion ... 74
image animation ... 178
image cache ... 178
image descriptor ... 169
image formats ... 168
image slice ... 177
image types ... 168
image-dired ... 601
image-dired mode ... 601
ImageMagick images ... 173
images in buffers ... 168
images, support for more formats ... 173
IMAP mailboxes ... 650
Imenu ... 427
in-situ subdirectory (Dired) ... 598
inbox file ... 635
incremental search ... 199
incremental search, input method interference ... 379
indentation ... 451, 483
indentation for comments ... 523
indentation for programs ... 517
index of buffer definitions ... 516
indicators, fringe ... 157
indirect buffers ... 286
inferior process ... 534
inheritance, text property ... 500
init file ... 711
init file '`.emacs`' on MS-Windows ... 785
init file, and non-ASCII characters ... 376
init file, not loading ... 757
initial options (command line) ... 754
input events ... 23
input focus ... 358
input methods ... 378
input methods, X ... 771

insert file contents, command-line argument ... 755
insert Unicode character ... 58
inserted subdirectory (Dired) ... 598
inserting blank lines ... 63
inserting killed text ... 466
inserting matching parentheses ... 523
insertion ... 58
insertion before point ... 458
insertion of text ... 458
`INSIDE_EMACS` environment variable ... 657
Integrated development environment ... 580
interactive call ... 17
interactive code description ... 14
interactive completion ... 14
interactive function ... 12
`interactive`, examples of using ... 17
`interactive-form`, symbol property ... 12
internal border width, command-line argument ... 766
internal windows ... 289
international characters in '`.emacs`' ... 716
international files from DOS/Windows systems ... 381
international scripts ... 374
interrupt Lisp functions ... 48
intervals ... 506
Intlfonts for PostScript printing ... 671
Intlfonts package, installation ... 389
invisible frame ... 360
invisible text ... 121
invisible/intangible text, and point ... 23
invocation (command line arguments) ... 754
invoking Emacs from Windows Explorer ... 781
IPA ... 374
isearch ... 199
Islamic calendar ... 611
ISO commercial calendar ... 611
ISO Latin character sets ... 393
`iso-ascii` library ... 393
`iso-transl` library ... 393
`ispell` program ... 221
italic text ... 139

## J

Japanese ... 374
Java mode ... 530
Javascript mode ... 514
JDB ... 539
Julian calendar ... 611
Julian day numbers ... 611
jumbled display of bidirectional text ... 197

## K

Kerberos POP authentication ... 650
key bindings ... 702
key rebinding, permanent ... 711
key rebinding, this session ... 705
key sequence input ... 39
keyboard events ... 23
keyboard events in strings ... 37
keyboard input ... 38
keyboard macro ... 222
keyboard macro execution ... 18
keyboard macro termination ... 194
keyboard macro, terminating ... 46
keyboard macros ... 54
keyboard shortcuts ... 810
keyboard, MS-Windows ... 785
keymap ... 703
keymap of character ... 495
keymap of character (and overlays) ... 134
keymaps in modes ... 400
keypad ... 708
keys in documentation strings ... 82
kill command repetition ... 21
kill DOS application ... 786
kill ring ... 97, 464
killing buffers ... 284
killing characters and lines ... 61
killing Emacs ... 57
killing expressions ... 521
killing rectangular areas of text ... 103
killing text ... 95
Korean ... 374

## L

label (Rmail) ... 638
landmark game ... 680
language environments ... 377
`language-change` event ... 32
Lao ... 374
largest window ... 309
Latin ... 374
lax space matching ... 201
layout on display, and bidirectional text ... 197
layout parameters of frames ... 349
lazy search highlighting ... 200
least recently used window ... 308
leaving Emacs ... 57
libraries ... 549
Life ... 680
line endings ... 382
line height ... 136
line mode (terminal emulator) ... 663
line number commands ... 64
line spacing ... 770
line spacing, command-line argument ... 767
line truncation ... 112
line truncation, and fringes ... 63
line wrapping ... 63, 112
linking files ... 248
links (customization buffer) ... 686
Lisp character syntax ... 713

Lisp editing ... 514
Lisp files byte-compiled by XEmacs ... 550
Lisp files, and multibyte operation ... 376
Lisp mode ... 552
Lisp object syntax ... 713
Lisp package ... 681
Lisp string syntax ... 712
Lisp symbol completion ... 528
`lisp-indent-function` property ... 519
'`lisp-mode.el`' ... 412
list commands ... 522
live buffer ... 285
live windows ... 289
ln ... 249
load init file of another user ... 757
load path for Emacs Lisp ... 549
loading Lisp code ... 549
loading Lisp libraries automatically ... 714
loading Lisp libraries, command-line argument ... 755
loading several files (in Dired) ... 595
local keymap ... 704
local variables ... 697
local variables in files ... 698
local variables, for all files in a directory ... 701
locales ... 377
locate file in path ... 247
location of point ... 64
lock file ... 239
locking-based version ... 556
locus ... 535
Log Edit mode ... 560
log File, types of ... 557
logging echo-area messages ... 117
logging keystrokes ... 726
logical order ... 195, 394
lowering a frame ... 361
lpr usage under MS-DOS ... 788
LRM ... 395
`ls` emulation ... 784

# M

`M-x` ... 19
M4 mode ... 514
Mac OS X ... 778
Macintosh ... 778
Macintosh end-of-line conversion ... 382
Macintosh key bindings ... 676
macro expansion in C ... 532
magic file names ... 262
mail ... 624
mail aliases ... 626
`MAIL` environment variable ... 635
Mail mode ... 631
mail signature ... 630
mail-composition methods ... 631
Mailclient ... 628
`MAILHOST` environment variable ... 649
mailrc file ... 626
main border width, command-line argument ... 766
major mode ... 399
major mode command ... 399
major mode conventions ... 399
major mode hook ... 402
major mode, automatic selection ... 403
make ... 534
`make-frame-visible` event ... 31
Makefile mode ... 514
making buttons ... 181
man page ... 526
man pages, and local file variables ... 699
manipulating buttons ... 181
manual pages, on MS-DOS/MS-Windows ... 527
Marathi ... 374
margins, display ... 167
mark ... 89
mark rectangle ... 103
mark ring ... 92
marker argument ... 16
marking executable files (in Dired) ... 592
marking many files (in Dired) ... 591
marking sections of text ... 90
marking subdirectories (in Dired) ... 592
marking symbolic links (in Dired) ... 592
matching parentheses ... 522
matching parenthesis and braces, moving to ... 522
maximized, command-line argument ... 765
maximizing windows ... 297
Mayan calendar ... 611
Mayan calendar round ... 614
Mayan haab calendar ... 614
Mayan long count ... 614
Mayan tzolkin calendar ... 614
MD5 checksum ... 510
memory full ... 720
menu bar ... 9, 771
menu bar access using keyboard (MS-Windows) ... 785
`menu-bar` prefix key ... 40
Mercurial ... 555
merging changes ... 569
merging-based version ... 556
message ... 624
message digest ... 510
Message mode ... 627
Message mode for sending mail ... 631
message number ... 632
messages saved from echo area ... 7
meta character printing ... 84
Metafont mode ... 514
MH mail interface ... 631
Microsoft Office file ... 653
Microsoft Windows ... 781
MIME ... 630
MIME messages (Rmail) ... 646

minibuffer ..... 7, 68
minibuffer confirmation ..... 72
Minibuffer Electric Default mode ..... 68
minibuffer history ..... 74
minibuffer history, searching ..... 203
minibuffer input ..... 51
minibuffer keymaps ..... 704
minibuffer window, and `next-window` ..... 307
minibuffer-only frame ..... 346
minimized frame ..... 360
minimizing ..... 57
minimizing a frame at startup ..... 767
minimizing windows ..... 297
minor mode ..... 413
minor mode conventions ..... 413
minor mode keymap ..... 704
mistakes, correcting ..... 217
mode ..... 396
mode help ..... 405
mode hook ..... 402, 514
mode line ..... 8, 418
mode line construct ..... 419
mode loading ..... 403
mode variable ..... 413
mode, AWK ..... 530
mode, C ..... 530
mode, C++ ..... 530
mode, Comint ..... 659
mode, Compilation ..... 535
mode, CORBA IDL ..... 530
mode, CRiSP ..... 676
mode, Delete Selection ..... 92
mode, Dirtrack ..... 662
mode, DocView ..... 653
mode, Electric Indent ..... 453
mode, Emacs Lisp ..... 550
mode, Flyspell ..... 221
mode, Goto Address ..... 678
mode, GUD Tooltip ..... 540
mode, Hexl ..... 674
mode, Hideshow ..... 527
mode, Java ..... 530
mode, Lisp ..... 552
mode, Log Edit ..... 560
mode, Mail ..... 631
mode, Message ..... 627
mode, Minibuffer Electric Default ..... 68
mode, Objective C ..... 530
mode, Occur ..... 215
mode, Occur Edit ..... 215
mode, Pike ..... 530
mode, Scheme ..... 552
mode, Semantic ..... 529
mode, Shell ..... 657
mode, Term ..... 664
mode, Transient Mark ..... 94
`mode-line` prefix key ..... 40
model/view/controller ..... 183
modes for programming languages ..... 514
modification flag (of buffer) ..... 278
modification time of buffer ..... 279
modification time of file ..... 245
modifier bits (of input character) ..... 23
modifier keys ..... 707
Modula2 mode ..... 514
moon, phases of ..... 610
Morse code ..... 680
most recently selected windows ..... 306
Motif key bindings ..... 676
motion event ..... 30
mouse button events ..... 709
mouse click event ..... 26
mouse drag event ..... 28
mouse events, data in ..... 35
mouse events, in special parts of frame ..... 40
mouse events, repeated ..... 29
mouse motion events ..... 30
mouse pointer color, command-line argument ..... 764
mouse pointer shape ..... 365
mouse position ..... 362
mouse position list ..... 26
mouse position list, accessing ..... 35
mouse tracking ..... 362
mouse, and MS-Windows ..... 785
mouse, availability ..... 371
mouse-1 ..... 502
move to beginning or end of function ..... 516
movemail ..... 649
`movemail` program ..... 648
movement ..... 59
moving files (in Dired) ..... 594
moving inside the calendar ..... 604
moving point ..... 59
moving text ..... 97
moving the cursor ..... 59
MS-DOS and file modes ..... 244
MS-DOS end-of-line conversion ..... 382
MS-Windows file-name syntax ..... 251
MS-Windows keyboard shortcuts ..... 785
MS-Windows, and primary selection ..... 101
MS-Windows, Emacs peculiarities ..... 781
MULE ..... 374
multi-file package ..... 683
multi-tty ..... 342
multibyte characters ..... 374
multibyte operation, and Lisp files ..... 376
multiline font lock ..... 438
multiple terminals ..... 342
multiple windows ..... 289
multiple X displays ..... 342
multiple-file search and replace ..... 578
Multipurpose Internet Mail Extensions ..... 630
Multithreaded debugging in GDB ..... 548

## N

'`net use`', and printing on MS-Windows ..... 787
networked printers (MS-Windows) ..... 787
new file message ..... 233
newline ..... 58
newline and Auto Fill mode ..... 460
newsreader ..... 651
Next Error Follow mode ..... 537
next input ..... 45
NFS and quitting ..... 717
nil ..... 814
`no-conversion`, coding system ..... 383
`no-self-insert` property ..... 582
non-ASCII characters in '`.emacs`' ..... 716
non-ASCII keys, binding ..... 716
non-greedy regexp matching ..... 207
Non-stop debugging in GDB ..... 548
nondirectory part (of file name) ..... 251
nonincremental search ..... 203
nonprinting characters, reading ..... 44
normal hook ..... 396, 696
NSA ..... 631
numeric arguments ..... 65
numeric prefix argument ..... 49
numeric prefix argument usage ..... 16
numerical RGB color specification ..... 367

## O

Objective C mode ..... 530
Occur Edit mode ..... 215
Occur mode ..... 215
octal character input ..... 44
octal escapes ..... 189
Octave mode ..... 514
opacity, frame ..... 353
open-parenthesis in leftmost column ..... 515
OpenDocument file ..... 653
operating on files in Dired ..... 593
operations on a marked region ..... 91
options (command line) ..... 754
ordering of windows, cyclic ..... 307
other editors ..... 676
out of memory ..... 720
overlays ..... 128
overlined text ..... 139
override character terminal color support ..... 764

## P

package ..... 681
package archive ..... 684
package attributes ..... 681
package autoloads ..... 681
package dependencies ..... 681
package name ..... 681
package version ..... 681
paging in Term mode ..... 664
paragraph, base direction ..... 395
parent window ..... 290, 291
parentheses, displaying matches ..... 522
parentheses, moving across ..... 522
parenthesis matching ..... 188
parenthetical groupings ..... 522
parsing html ..... 510
parsing xml ..... 511
paste ..... 100
pasting ..... 97
patches, sending ..... 730
PBM ..... 174
PC key bindings ..... 676
PC selection ..... 676
PDB ..... 539
PDF file ..... 653
peeking at input ..... 45
per-buffer variables ..... 697
per-directory local variables ..... 701
percent symbol in mode line ..... 419
Perl mode ..... 514
Perldb ..... 539
permissions, file ..... 244, 249
Persian calendar ..... 612
phases of the moon ..... 610
Pike mode ..... 530
point ..... 6
point in window ..... 323
point location ..... 64
point location, on MS-DOS ..... 782
pointer shape ..... 365
Polish ..... 374
Pong game ..... 680
POP mailboxes ..... 649
position and size of Emacs frame ..... 765
position argument ..... 15
position in window ..... 323
position of mouse ..... 362
PostScript file ..... 653
postscript images ..... 173
PostScript mode ..... 514
prefix argument ..... 49
prefix argument unreading ..... 45
prefix arguments ..... 65
preprocessor highlighting ..... 532
pretty-printer ..... 517
preventing quitting ..... 48
primary Rmail file ..... 632
primary selection ..... 90, 101, 366
printable ASCII characters ..... 189
printing ..... 669
printing files (in Dired) ..... 595
Printing package ..... 672
Prog mode ..... 697
`prog-mode`, and `bidi-paragraph-direction` .. 196
program building ..... 534
program editing ..... 514, 697
progress reporting ..... 115

Prolog mode ... 514
prompt ... 68
prompt, shell ... 659
properties of text ... 489
property category of text character ... 494
PS file ... 653
puzzles ... 679
Python mode ... 514

## Q

query replace ... 213
quitting ... 48, 717
quitting (in search) ... 201
quitting Emacs ... 57
quoted character input ... 44
quoting ... 58

## R

raising a frame ... 361
raw prefix argument ... 49
raw prefix argument usage ... 16
`raw-text`, coding system ... 382
RCS ... 555
read command name ... 19
read input ... 38
read-only buffer ... 280
read-only buffers in interactive ... 13
read-only character ... 496
read-only text, killing ... 97
`read-quoted-char` quitting ... 48
reading a single event ... 41
reading from files ... 236
reading interactive arguments ... 14
reading mail ... 632
reading order ... 195
rebinding keys, permanently ... 711
rebinding major mode keys ... 706
rebinding mouse buttons ... 709
rebinding non-ASCII keys ... 716
record command history ... 18
rectangle ... 103
rectangle highlighting ... 105
recursive command loop ... 51
recursive copying ... 594
recursive deletion ... 590
recursive editing level ... 51, 675
redefining keys, this session ... 705
redo ... 217, 469
refreshing displayed files ... 599
regexp ... 206
regexp search ... 205
region ... 89
region argument ... 16
region highlighting ... 94
registered file ... 555
registers ... 106, 507
registry, setting environment variables (MS-Windows) ... 762
registry, setting resources (MS-Windows) ... 769
regular expression ... 206
related files ... 533
relative file name ... 253
reload files ... 675
remember editing session ... 675
remote host ... 664
remote host, debugging on ... 540
remove indentation ... 452
renaming files (in Dired) ... 594
repeat events ... 29
repeating a command ... 66
replace characters ... 507
replacement ... 211
reply to a message ... 640
reporting bugs ... 725
repository ... 555
resize frame ... 354
resize window ... 295
resizing minibuffer ... 69
resolving conflicts ... 569
resource files for GTK ... 771
resources ... 769
restore session ... 675
resume (cf. `no-redraw-on-reenter`) ... 111
retrying a failed message ... 641
reverse order in POP inboxes ... 650
reverse video, command-line argument ... 764
revision ... 555
revision ID ... 555
revision ID in version control ... 560
rgb value ... 368
right-to-left text ... 195, 394
risky variable ... 701
RLM ... 395
Rlogin ... 664
rm ... 249
Rmail ... 632
Rmail file sorting ... 645
Romanian ... 374
root window ... 290
rot13 code ... 648
Ruby mode ... 514
runemacs.exe ... 781
running a hook ... 696
running Lisp functions ... 534

## S

saved echo area messages ... 7
saving a setting ... 689
saving buffers ... 234
saving file name in a register ... 108
saving keyboard macros ... 227
saving number in a register ... 108
saving position in a register ... 106

saving rectangle in a register ... 107
saving sessions ... 675
saving text in a register ... 106
saving text properties ... 267
saving window configuration in a register ... 107
saving window information ... 335
scalability of overlays ... 128
SCCS ... 555
Scheme mode ... 552
screen ... 6
screen of terminal ... 289
screen reader software, MS-Windows ... 790
screen size ... 354
script mode ... 756
scroll bar events, data in ... 37
scroll bars ... 162
`scroll-command` property ... 329
scrolling all windows ... 676
scrolling in the calendar ... 606
scrolling textually ... 327
SDB ... 539
search and replace in multiple files ... 578
search and replace in multiple files (in Dired) ... 595
search for a regular expression ... 205
search multiple files (in Dired) ... 595
search ring ... 200
search-and-replace commands ... 211
searching ... 199
searching Dired buffers ... 589
searching in Rmail ... 634
searching multiple files via Dired ... 602
secondary selection ... 101, 366
sections of manual pages ... 526
select all ... 91
selected window ... 290
selecting a buffer ... 272
selecting a window ... 306
selection (for window systems) ... 366
selection, primary ... 101
selective undo ... 217
`self-insert-command`, minor modes ... 415
self-insertion ... 460
SELinux context ... 246
Semantic mode ... 529
Semantic package ... 529
sending mail ... 624
sending patches for GNU Emacs ... 730
Sendmail ... 628
server file ... 668
server, using Emacs as ... 665
setting a mark ... 89
setting modes of files ... 248
setting variables ... 695
settings ... 686
settings, how to save ... 689
severity level ... 118
sexp ... 521
SHA hash ... 510
shell commands ... 655
shell commands, Dired ... 595
`SHELL` environment variable ... 656
Shell mode ... 657
shell scripts, and local file variables ... 699
Shell-script mode ... 514
shelves in version control ... 567
shift-selection ... 90, 93
shift-selection, and `interactive` spec ... 13
shift-translation ... 40
Show Paren mode ... 523
showing hidden subdirectories (Dired) ... 599
shy group, in regexp ... 209
sibling window ... 290
signing files (in Dired) ... 595
`sigusr1` event ... 31
`sigusr2` event ... 31
simple package ... 682
Simula mode ... 514
simulation of middle mouse button ... 785
single file package ... 682
site init file ... 711
site-lisp directories ... 712
'`site-start.el`' file, not loading ... 757
'`site-start.el`', the site startup file ... 711
size of frame ... 354
size of minibuffer ... 69
size of window ... 293
slashes repeated in file name ... 69
slice, image ... 177
Slovak ... 374
Slovenian ... 374
SMTP ... 628
Snake ... 680
solitaire ... 680
sorting ... 672
sorting Dired buffer ... 600
sorting Rmail file ... 645
sorting text ... 479
`space` display spec, and bidirectional text ... 197
spaces, pixel specification ... 165
spaces, specified height or width ... 164
Spanish ... 374
special events ... 46
`special` modes ... 402
specific version control system ... 560
specify color ... 367
specify default font from the command line ... 763
specify end-of-line conversion ... 386
specifying fullscreen for Emacs frame ... 765
spell-checking the active region ... 220
spelling, checking and correcting ... 219
splash screen ... 757
splitting windows ... 297
standard colors for character terminals ... 352
standard colors on a character terminal ... 764
standard fontset ... 390
start directory, MS-Windows ... 781

start iconified, command-line argument ..... 767
starting Emacs ..... 56
starting Emacs on MS-Windows ..... 781
startup (command line arguments) ..... 754
startup (init file) ..... 711
startup fontset ..... 390
startup message ..... 757
startup screen ..... 56
stashes in version control ..... 567
sticky text properties ..... 500
strike-through text ..... 139
string substitution ..... 211
string syntax ..... 712
string, writing a doc string ..... 79
strings with keyboard events ..... 37
style (for indentation) ..... 519
subdirectories in Dired ..... 598
subprocesses on MS-Windows ..... 786
subscribe groups ..... 652
subshell ..... 655
substituting keys in documentation ..... 82
Subversion ..... 555
summary (Rmail) ..... 642
summing time intervals ..... 622
sunrise and sunset ..... 609
suspend (cf. `no-redraw-on-reenter`) ..... 111
suspend evaluation ..... 52
suspending ..... 57
suspicious constructions in C, C++ ..... 533
SVN ..... 555
swap text between buffers ..... 287
switches (command line) ..... 754
switching to a buffer ..... 311
symbol search ..... 204
symbolic links (creation in Dired) ..... 594
syntactic font lock ..... 437
syntax of regexps ..... 206
syntax table example ..... 412
syntax tables in modes ..... 401
system abbrev ..... 581

## T

t ..... 814
tab deletion ..... 461
tab stops ..... 452
tabs ..... 451
tabs stops for indentation ..... 487
Tabulated List mode ..... 409
tags and tag tables ..... 571
tags, C++ ..... 572
tags-based completion ..... 528
Tcl mode ..... 514
Telnet ..... 664
`TEMP` environment variable ..... 257
`TERM` environment variable ..... 726
Term mode ..... 664
terminal ..... 341
terminal parameters ..... 355
terminal screen ..... 289
terminal, serial ..... 665
terminal-local variables ..... 343
termscript file ..... 726
Tetris ..... 680
`TEXEDIT` environment variable ..... 665
text ..... 454
text and binary files on MS-DOS/MS-Windows ..... 782
text area of a window ..... 293
text colors, from command line ..... 763
text deletion ..... 460
text insertion ..... 458
text near point ..... 454
text properties ..... 489
text properties at point ..... 375
text properties in files ..... 267
text properties in the mode line ..... 425
text terminal ..... 341
textual scrolling ..... 327
Thai ..... 374
`throw` example ..... 52
Tibetan ..... 374
TIFF ..... 173
tiled windows ..... 289
time intervals, summing ..... 622
timeclock ..... 622
timestamp of a mouse event ..... 36
TLS encryption (Rmail) ..... 650
`TMP` environment variable ..... 257
`TMPDIR` environment variable ..... 257
toggling marks (in Dired) ..... 592
tool bar ..... 771
tooltip ..... 495
top frame ..... 361
top level ..... 8
total height of a window ..... 294
total width of a window ..... 294
tower of Hanoi ..... 679
TPU ..... 677
Transient Mark mode ..... 94
transparency, frame ..... 353
transposition of expressions ..... 521
trash ..... 249, 262
triple clicks ..... 709
triple-click events ..... 29
truename (of file) ..... 243
truncation ..... 63
Turkish ..... 374
turn multibyte support on or off ..... 376
two directories (in Dired) ..... 594
types of log file ..... 557
typos, fixing ..... 217

## U

umask ..... 250

unbalanced parentheses and quotes ..... 520
`undecided`, coding system ..... 382
undeletion (Rmail) ..... 635
underlined text ..... 139
undigestify ..... 648
undisplayable characters ..... 375
undo ..... 217
undo avoidance ..... 507
undo limit ..... 218
unibyte buffers, and bidi reordering ..... 196
Unibyte operation ..... 393
unibyte operation, and Lisp files ..... 376
Unicode ..... 374
unicode bidirectional algorithm ..... 195
Unicode characters, inserting ..... 58
unmarking files (in Dired) ..... 592
unsubscribe groups ..... 652
untranslated file system ..... 783
upcase file names ..... 597
updating Dired buffer ..... 599
upper case key sequence ..... 40
URLs ..... 677
URLs, activating ..... 678
Usenet news ..... 651
user option ..... 686
user options, changing ..... 688
user signals ..... 31
UTF-8 ..... 377

## V

valid windows ..... 289
variable ..... 694
variable-width spaces ..... 164
variables, changing ..... 688
VC Directory buffer ..... 565
VC filesets ..... 558
VC mode line indicator ..... 557
verifying digital signatures on files (in Dired) .. 595
version control ..... 554
version control log ..... 557
version control status ..... 557
version number (in file name) ..... 251
vertical combination ..... 291
vertical fractional scrolling ..... 330
vertical scroll bars, command-line argument ... 767
vertical scroll position ..... 330
`vertical-line` prefix key ..... 40
`vertical-scroll-bar` prefix key ..... 40
VHDL mode ..... 514
vi ..... 676
Vietnamese ..... 374
view part, model/view/controller ..... 183
virtual buffers ..... 287
visible frame ..... 360
visited file ..... 276
visited file mode ..... 403
visiting files ..... 230
visiting files, command-line argument ..... 754
visual order ..... 195, 394

## W

waiting ..... 47
waiting for command key input ..... 45
warning type ..... 119
warnings ..... 118
Watching expressions in GDB ..... 547
wdired mode ..... 601
Web ..... 677
weeks, which day they start on ..... 605
`wheel-down` event ..... 31
`wheel-up` event ..... 31
whitespace character ..... 451
width and height of Emacs frame ..... 765
width of a window ..... 294
window ..... 289
window body ..... 293
window body height ..... 294
window body size ..... 294
window body width ..... 294
window combination ..... 291
window combination limit ..... 304
window configurations ..... 335
window end position ..... 324
window header line ..... 426
window height ..... 294
window history ..... 319
window in direction ..... 292
window manager interaction, and frame parameters ..... 350
window ordering, cyclic ..... 307
window parameters ..... 337
window point ..... 323
window position ..... 323, 333
window position on display ..... 347
window positions and window managers ..... 348
window resizing ..... 295
window selected within a frame ..... 290
window size ..... 293
window size on display ..... 348
window size, changing ..... 295
window splitting ..... 297
window start position ..... 324
window that satisfies a predicate ..... 309
window top line ..... 324
window tree ..... 290
window width ..... 294
Windows system menu ..... 785
windows, controlling precisely ..... 309
word search ..... 203
word wrap ..... 64
WordStar ..... 676
work file ..... 555
working tree ..... 555
World Wide Web ..... 677

wrapping ..... 63
writing a documentation string ..... 79
writing to files ..... 237

## X

X cutting and pasting ..... 101
X input methods ..... 771
X resources ..... 769
X selection ..... 101
X Window System ..... 194
XBM ..... 172
XDB ..... 539
XIM ..... 771
XPM ..... 172

## Y

y or n prompt ..... 77
yahrzeits ..... 613
yanking ..... 97
yanking and text properties ..... 466
yanking previous kills ..... 98
yes or no prompt ..... 77

## Z

Zmacs mode ..... 94